EMBARCADERO TECHNOLOGIES®

White Paper

The New DataSnap in Delphi 2009

Building Multi-Tier Applications without Using a COM Interface

By Marco Cantù

December 2008

INTRODUCTION

For a long time Delphi has included a technology for building multi-tier database applications. Formerly known as MIDAS and later as DataSnap, Delphi's multi-tier technology was based on COM, even if the remote connectivity could be provided by sockets and HTTP, instead of DCOM. For some time, it even supported CORBA--a slightly modified version that provided SOAP connectivity.

Delphi 2009 still includes the classic DataSnap, but provides a new remoting and multi-tier technology as well. It is partially based on the dbExpress architecture. This new technology is still called DataSnap, but to avoid confusion is generally referenced as “DataSnap 2009”.

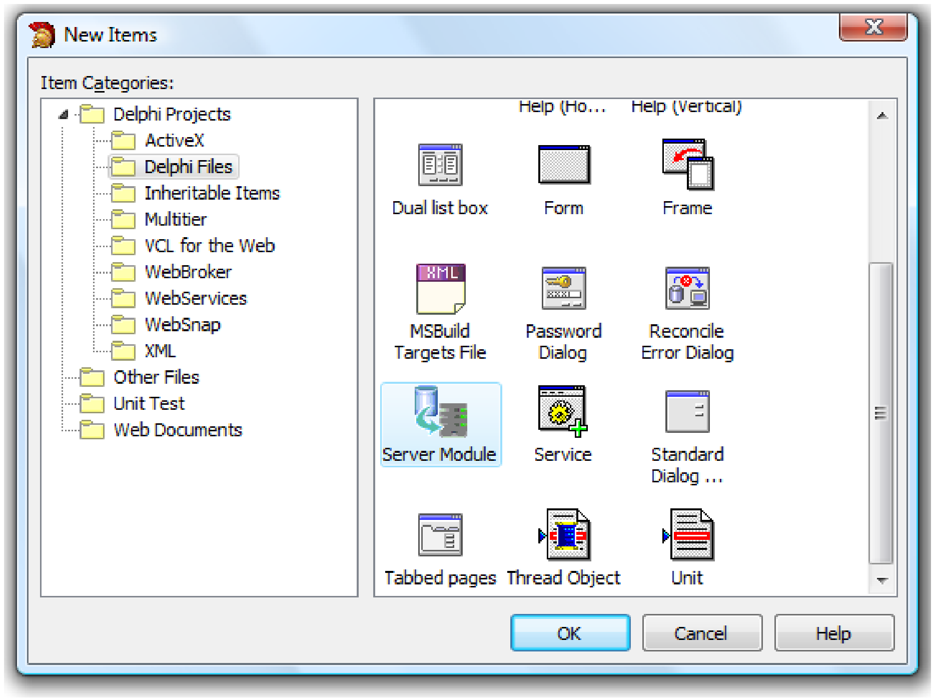
BUILDING A DATASNAP 2009 DEMO

Before I get into too many details, let me start with a simple three-tier database-oriented demo. This will help clarify a few points and also cover differences from the previous version.

BUILDING A SERVER

The first step is building a DataSnap 2009 server application. This can be a standard VCL application, to which you add a server module (found in the Delphi Files page of the New Items dialog box and not in the Multitier page):

Figure 1 Creating a Server Module in Delphi 2009



To the server module (but we could also have used a standard data module) you generally add the dbExpress components to connect to the database server, plus a dataset provider to expose the given datasets:

```
object IBCONNECTION: TSQLConnection
  ConnectionName = 'IBCONNECTION'
  DriverName = 'Interbase'
  LoginPrompt = False
  Params.Strings = (
    'DriverName=Interbase'
    'Database=C:\Program Files\...\Data\Employee.GDB')
end
object EMPLOYEE: TSQLDataSet
  CommandText = 'EMPLOYEE'
  CommandType = ctTable
  SQLConnection = IBCONNECTION
end
object DataSetProviderEmployee: TDataSetProvider
  DataSet = EMPLOYEE
end
```

This server module is built in a very similar way as it has been in the past. What is new is the need to include three new components that provide configuration and connectivity in place of the COM support (which is totally gone). The three components are:

- DSServer, the main server configuration component, which is needed to wire all the other DataSnap 2009 components together.
- DSServerClass, a component needed for each class you want to expose. This component is not the class you make available, but acts as a class factory to create objects of the class you want to call from a remote client. In other words, the DSServerClass component will refer to the class that has the public interface.
- DSTCPServerTransport, a component that defines the transport protocol to be used (this is the only protocol directly available in Delphi 2009) and its configuration, such as which TCP/IP port to use.

In the demo these components are in the main form of the server, configured as follows:

```
object DSServer1: TDSServer
  AutoStart = True
  HideDSAdmin = False
  OnConnect = DSServer1Connect
  OnDisconnect = DSServer1Disconnect
end
object DSTCPServerTransport1: TDSTCPServerTransport
  PoolSize = 0
  Server = DSServer1
  BufferKBSize = 32
end
object DSServerClass1: TDSServerClass
  OnGetClass = DSServerClass1GetClass
  Server = DSServer1
  LifeCycle = 'Session'
end
```

We'll get to some of the details of these properties later on. The reason you don't see the value of the TCP/IP port in the listing above is that I've not modified the default value of 211.
The only Delphi code you need to write is the "class factory" code that you need to connect the `DSServerClass1` component to the server module exposing the providers:

```
procedure TFormFirst3Tier2009Server.
  DSServerClass1GetClass(DSServerClass: TDSServerClass;
  var PersistentClass: TPersistentClass);
begin
  PersistentClass := TDSFirst3TierServerModule;
end;
```

This is all you need for the server. I added a logging statement to the method above, as well as to the event handlers of the `OnConnect` and `OnDisconnect` events of the DSServer component.

Again, there is no need to register it in any way. Simply run it, maybe using the *Run | Run Without Debugging* command in the Delphi IDE, so you can build the client and connect it to the server even at design time.

THE FIRST CLIENT

Now that we have a server available, we can move on and build the first client. In the DataSnap 2009 client application, we need to use an SQLConnection component associated with the new DataSnap dbExpress driver and configured with the proper TCP/IP port.

Next we need a DSProviderConnection component, used to refer to the server class, with the `ServerClassName` property. This is not the intermediary class factory in the server (`DSServerClass1`), but the actual target of the class factory. In my example, it is the `TDSFirst3TierServerModule` class.

As with a traditional DataSnap application, the ClientDataSet component can use the provider to fetch (and update) the remote dataset. First, you have to assign the `RemoteServer` property of the ClientDataSet, picking the `DSProviderConnection1` component from the drop-down list. Next, you can select the `DataSetProviderEmployee` provider from the drop down of the `ProviderName` property, populated with all exported DataSetProvider components of the remote data module.

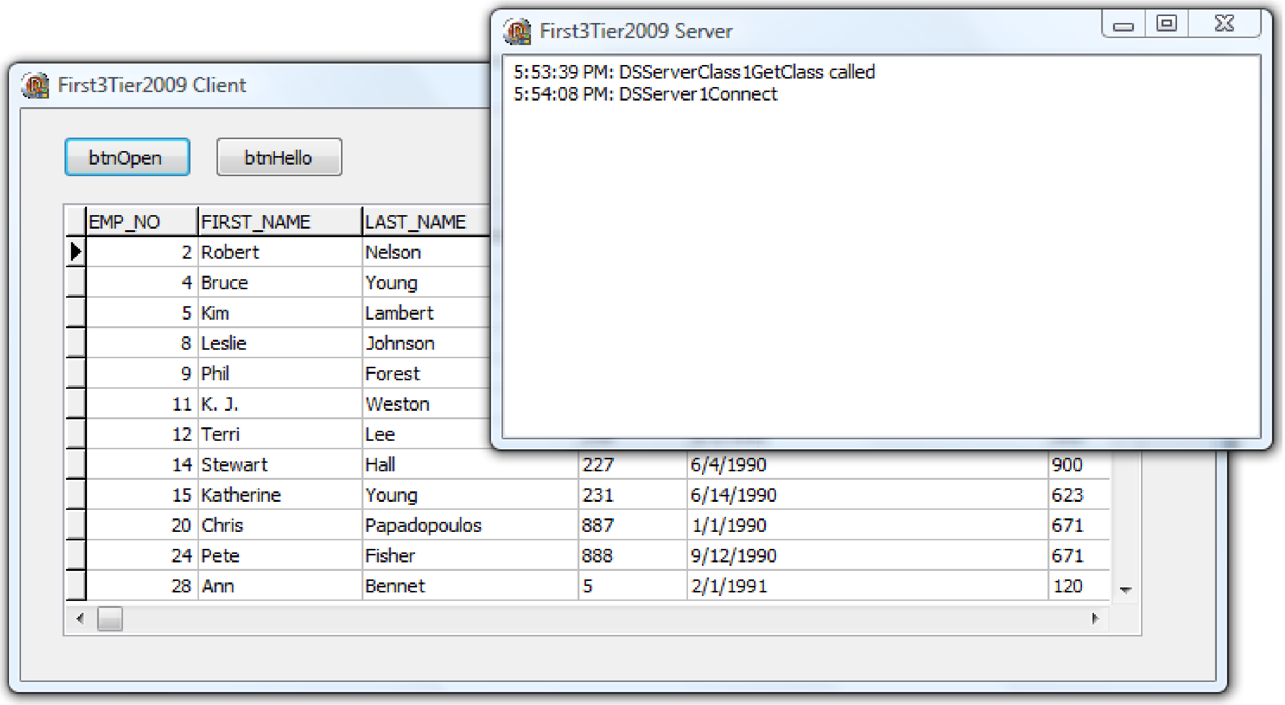
This is a summary of the properties of these components, including a DataSource used to display the database table in a DBGrid:

```
object SQLConnection1: TSQLConnection
  DriverName = 'Datasnap'
end
object DSProviderConnection1: TDSProviderConnection
  ServerClassName = 'TDSFirst3TierServerModule'
  SQLConnection = SQLConnection1
end
object ClientDataSet1: TClientDataSet
  ProviderName = 'DataSetProviderEmployee'
  RemoteServer = DSProviderConnection1
```

```
end
object DataSource1: TDataSource
  DataSet = ClientDataSet1
end
```

That's all it takes for an introductory demo. Now if you run the server first and the client next, you can press the Open button of the client and see the database data. Also notice the log produced by the server, shown in Figure 2:

Figure 2 View Database Data as Well as Server Logs



```
5:53:39 PM: DSServerClass1GetClass called
5:54:08 PM: DSServer1Connect
```

EMP_NO	FIRST_NAME	LAST_NAME			
2	Robert	Nelson			
4	Bruce	Young			
5	Kim	Lambert			
8	Leslie	Johnson			
9	Phil	Forest			
11	K. J.	Weston			
12	Terri	Lee			
14	Stewart	Hall	227	6/4/1990	900
15	Katherine	Young	231	6/14/1990	623
20	Chris	Papadopoulos	887	1/1/1990	671
24	Pete	Fisher	888	9/12/1990	671
28	Ann	Bennet	5	2/1/1991	120

From DataSnap to DataSnap 2009

Compared to the traditional DataSnap application, there are a few significant differences, more related to the architecture and deployment than the actual code you have to write:

- There is no COM involved for the development of the server. Even if a client could already use sockets in the past, a socket-to-Com mapping service was required on the server. Now the client and server applications communicate directly over TCP/IP.
- As a side effect, you don't have to register the server, nor run any helper service on it. All the server has to provide to the client is an open TCP/IP port the client can reach
- You must manually run the application on the server, or create a service for it. In the past the COM support implied the server application would be started as needed.
- The server implementation is slightly more complicated in terms of components, but there is very little code behind the scenes, as for the COM counterpart.

- The client implementation is almost identical, as we need a standard SQLConnection component, in place of a specific connection object.
- On the server side, the `TDSServerModule` class inherits from `TDataModule`, including the `IAppServer` interface (the same interface used in the past by a COM-based `TRemoteDataModule`) and enabling the `$MethodInfo` compiler directive.
- As the client-side dbExpress driver is a pure 100% Delphi driver, you don't need do deploy any DLL on the client computer, even if you are using dbExpress for the connectivity.

Pay a lot of attention when closing the server application. Unlike in the COM architecture, which warns you about pending connections, a DataSnap 2009 server will seem to close, but won't until there are no remaining connections to it. However, even after the connections have been closed it will remain running in memory, even if the main form is gone. You'll need to use Task Manager (or Process Explorer) to terminate the server. You might think that closing all existing client applications will be enough, but it is not: The Delphi IDE, in fact, can open a connection to the server even automatically, for browsing its exposed classes and methods. Be sure to close any SQLConnection to the server before stopping it.

ADDING SERVER METHODS

As in the past, you can write methods in the server that can be called by the client. These were based on COM, so you had to add interfaces to the type library and implement them in the server objects, and call the methods using COM dispatch interfaces on the client. In DataSnap 2009 the remote methods calls, or server method calls, are based on Delphi's RTTI. Notice, however, that parameters passing is based on dbExpress parameter types, and not on Delphi language types.

You can have multiple server side classes that expose methods, but to continue with the simple project I've already built, I added an extra method to the server module class (in the server application), using the following code:

```
type
  TDSFirst3TierServerModule = class(TDSServerModule)
    IBCONNECTION: TSQLConnection;
    EMPLOYEE: TSQLDataSet;
    DataSetProviderEmployee: TDataSetProvider;
  private
    { Private declarations }
  public
    function GetHello: string;
  end;

function TDSFirst3TierServerModule.GetHello: string;
begin
  Result := 'Hello from TDSFirst3TierServerModule at '
    + TimeToStr (Now);
end;
```

To enable remote invocation you have to connect the class for which you want to expose methods to a DSServerClass factory. (In this case, we've already done so in the database portion of the demo). The second requirement is to use a class that is compiled with the `$MethodInfo` directive turned on, but this already takes place in the declaration of the base

`TDSServerModule` class. This means that, in practice, all we have to do is to add a public method to the server module, and everything else will work.

How do we call this server method from the client application? There are basically two alternatives. One is to use the new SqlServerMethod component and call the server method as if it was a stored procedure. The second is to generate a proxy class in the client application and use this proxy class to make the call.

In the following client demo I've implemented both approaches. For the first, I've added an SqlServerMethod component to the form of the client, tied it to the connection, picked a value for the `ServerMethodName` property in the Object Inspector (among the many available, as the standard `IAppServer` interface methods are listed as well), and checked the value of the `Params` property. This is a copy of the component settings (which actually include the result of a sample call performed when checking the parameters):

```
object SqlServerMethod1: TSqlServerMethod
  GetMetadata = False
  Params = <
    item
      DataType = ftWideString
      Precision = 2000
      Name = 'ReturnParameter'
      ParamType = ptResult
      Size = 2000
      Value = 'Hello from TDSFirst3TierServerModule...'
    end>
  SQLConnection = SQLConnection1
  ServerMethodName = 'TDSFirst3TierServerModule.GetHello'
end
```

The native string type is mapped to a string parameter of 2,000 characters. After configuring the SqlServerMethod component, the program can call it using the input parameters (none in this case) and the output parameters (the result) as in a stored procedure or query call:

```
procedure TFormFirst3Tier2009Client.btnHelloClick(
  Sender: TObject);
begin
  SqlServerMethod1.ExecuteMethod;
  ShowMessage (SqlServerMethod1.Params[0].Value);
end;
```

To make it easier to write the calling code we can use the second approach I mentioned earlier, creating a local proxy class in the client application. To accomplish this, we can ask the Delphi IDE to parse the interface of the server class and create local proxy class for it, by clicking on the SQLConnection component and selecting the command *Generate Datasnap client classes*. In the case of this example, Delphi will generate a unit with the following class (from which I've omitted the code of the constructors and the destructor):

```
type
  TDSFirst3TierServerModuleClient = class
  private
    FDBXConnection: TDBXConnection;
    FInstanceOwner: Boolean;
    FGetHelloCommand: TDBXCommand;
  public
    constructor Create(
      ADBXConnection: TDBXConnection); overload;
    constructor Create(
      ADBXConnection: TDBXConnection;
      AInstanceOwner: Boolean); overload;
    destructor Destroy; override;
    function GetHello: string;
  end;

function TDSFirst3TierServerModuleClient.GetHello: string;
begin
  if FGetHelloCommand = nil then
  begin
    FGetHelloCommand := FDBXConnection.CreateCommand;
    FGetHelloCommand.CommandType :=
      TDBXCommandTypes.DSServerMethod;
    FGetHelloCommand.Text :=
      'TDSFirst3TierServerModule.GetHello';
    FGetHelloCommand.Prepare;
  end;

  FGetHelloCommand.ExecuteUpdate;
  Result := FGetHelloCommand.Parameters[0].
    Value.GetWideString;
end;
```

The generated code doesn't use the high level SqlServerMethod component, but rather calls directly into the low-level dbExpress implementation objects, like the `TDBXCommand` class.

Having this proxy class available, the client program can now call the server method in a more language-friendly way, although we do need to create an instance of the proxy class (or create one and keep it around). This code does exactly the same as the previous code based on the SqlServerMethod component:

```
procedure TFormFirst3Tier2009Client.btnHelloClick(
  Sender: TObject);
begin
  with TDSFirst3TierServerModuleClient.Create(
    SQLConnection1.DBXConnection) do
  try
    ShowMessage (GetHello);
  finally
    Free;
  end;
end;
```

If the code is actually longer than the previous version, this is because the method we are calling has no parameters, thus making the language binding code less relevant. Still, having a ready-to-use proxy object, we could have written:

```
  ShowMessage (ServerProxyObject.GetHello);
```

SESSIONS AND THREADING WITH A NON-DATABASE DATASNAP SERVER

If using the `IAppServer` interface directly is going to be the most common way for using DataSnap 2009, it is possible to use this multi-tier technology for remote method invocation outside of the database context. You can also use the same technology to access database data or perform database operations without using the `IAppServer` interface, which is fine if all you want to do is read data from the server. If you want to let the client application modify the data and post it back to the server, using custom methods could become tedious compared to the ready-to-use `IAppServer` interface, implemented by the ClientDataSet and the DataSetProvider components.

In any case, in this second example, I want to create a minimal server exposing a couple of simple classes. In the following sections I'll use this simple server to explore a couple of relevant issues, like server memory management and server (and client) threading.

The first server class (with two methods) I want to publish in the DsnapMethodServer project is the following:

```
{$MethodInfo ON}
type
  TSimpleServerClass = class(TPersistent)
  public
    function Echo (const Text: string): string;
    function SlowPrime (MaxValue: Integer): Integer;
  end;
{$MethodInfo OFF}
```

The code of the first method simply echoes the input, repeating its last part, while the second method performs the most classic slow computation. This is the code of the two methods:

```
function TSimpleServerClass.Echo(
  const Text: string): string;
begin
  Result := Text + '...' +
    Copy (Text, 2, maxint) + '...' +
    Copy (Text, Length (Text) - 1, 2);
end;

function TSimpleServerClass.SlowPrime(
  MaxValue: Integer): Integer;
var
  I: Integer;
begin
  // counts the prime numbers below the given value
  Result := 0;
  for I := 1 to MaxValue do
  begin
    if IsPrime (I) then
      Inc (Result);
  end;
```

```
end;
```

I've omitted the extra statements used to log the server operations from the code snippet above.

The server application has only one unit, which defines the main form and two server side classes. The form has the usual DataSnap server components, a DSServer and a DSTCPServerTransport, plus two DSServerClass component, one for each of the classes I want to expose. After compiling the server and starting it, I've let Delphi create a client proxy using the SQLConnection component of a new client application. This is the client proxy class:

```
type
  TSimpleServerClassClient = class
  private
    FDBXConnection: TDBXConnection;
    FInstanceOwner: Boolean;
    FEchoCommand: TDBXCommand;
    FSlowPrimeCommand: TDBXCommand;
  public
    constructor Create(
      ADBXConnection: TDBXConnection); overload;
    constructor Create(
      ADBXConnection: TDBXConnection;
      AInstanceOwner: Boolean); overload;
    destructor Destroy; override;
    function Echo(Text: string): string;
    function SlowPrime(MaxValue: Integer): Integer;
  end;
```

In the client program, the `OnClick` event of the button calls the `Echo` server method, after creating an instance of the proxy, if needed:

```
procedure TFormDsnapMethodsClient.btnEchoClick(
  Sender: TObject);
begin
  if not Assigned (SimpleServer) then
    SimpleServer := TSimpleServerClassClient.Create (
      SQLConnection1.DBXConnection);
  Edit1.Text := SimpleServer.Echo(Edit1.Text);
end;
```

In the example, pressing this button the sample text “Marco” is transformed by the server call into “Marco...arco...co”. This is a complete example of how you can create a totally custom server, with no database access involved and no use of the `IAppServer` interface. This is not the only method invocation technique available in Delphi, as you can use SOAP, socket-based applications, or third-party tools... but having this extra feature on top of the remote database access capability is certainly a plus.

One of the reasons I'm focusing on this example is it helps clarify some relevant features of DataSnap 2009. One of them is how server side objects relate to client proxies or server method invocation. This is better demonstrated by a server object that keeps track of its own state, like the following second server class of the demo project:

```
{$MethodInfo ON}
type
  TStorageServerClass = class(TPersistent)
  private
    FValue: Integer;
  public
    procedure SetValue(const Value: Integer);
    function GetValue: Integer;
    function ToString: string; override;
  published
    property Value: Integer read GetValue write SetValue;
  end;
{$MethodInfo OFF}
```

While the getter and setter methods simply read and write the local field, the `ToString` function returns both the value and an object identifier based on its hash code:

```
function TStorageServerClass.ToString: string;
begin
  Result := 'Value: ' + IntToStr (Value) +
     ' - Object: ' + IntToHex (GetHashCode, 4);
end;
```

I'll use this method to figure out how the life cycle of server objects work. In this class the property definition only makes sense for the server as it is not exposed to the client. The interface of the corresponding proxy becomes (after removing private fields, standard constructors and destructor):

```
type
  TStorageServerClassClient = class
  public
    procedure SetValue(Value: Integer);
    function GetValue: Integer;
    function ToString: string;
```

Notice that compiling this class produces the following warning, unless you manually mark the method as `override`:

```
Method 'ToString' hides virtual method of base type 'TObject'
```

The goal of this example is to figure out what happens when multiple client applications use the same server. The behavior of a DataSnap 2009 server in such a case depends on the value of the `LifeCycle` string property of the DSServerClass component being used.

Server Objects Life Cycle

The life cycle of DataSnap 2009 server objects depends on the corresponding setting of the related DSServerClass component. The `LifeCycle` property of this component can assume the following three string values (which are read from the DSServerClass components when the DSServer object is opened, ignoring any change at runtime):

- Session indicates that the server will create a different object for each client socket connection, that is, a server object for each client. The server objects are released when the connection is closed. Multiple clients will have independent status and separate database access in case the server object is a data module, maybe with its own database connection component. This is the default setting.
- Invocation indicates that a new server object is created (and destroyed) every time the server method is invoked. This is a classic stateless behavior, making the server extremely scalable, but also subject to fetching the same data over and over.
- Server indicates a shared server object, a singleton. Each client will use the same server object instance, the same data, potentially causing synchronization problems (as different client invocations are performed by different server threads). Access to shared server objects must be protected by synchronization techniques (for example using the new `TMonitor` record).

Besides using these default settings, you can customize the creation and destruction of server side objects using the `OnCreateInstance` and `OnDestroyInstance` events of the DSServerClass component. This can be used to implement server-side object pooling.

A Client Starting the Server and Opening Multiple Connections

As a practical example, the DsnapMethods project lets you create multiple client connections from a single instance of a client application (using multiple instances will yield the same result), You can create multiple instances of the form that has the SQLConnection component and store a local instance of the client proxy that is created the first time it is used. Not only can the client create multiple client connections, but it can also start the server program with a given life cycle setting. This is easy to achieve because the client and the server application are on the same computer.

To accomplish this I've added to the unit of the main form of the server a global variable, used to determine the DSServerClass `LifeCycle` property:

```
var
  ParamLifeCycle: string;

procedure TFormDsnapMethodsServer.DSServerClass2GetClass(
  DSServerClass: TDSServerClass;
  var PersistentClass: TPersistentClass);
begin
  DSServerClass2.LifeCycle := ParamLifeCycle;
  Log ('LifeCycle: ' + DSServerClass2.LifeCycle);
  PersistentClass := TStorageServerClass;
end;
```

The value of the `ParamLifeCycle` global variable is initialized using the command line parameters of the server application, which has the following code at the beginning of its project file source code:

```
begin
  if ParamCount > 0 then
    ParamLifeCycle := ParamStr(1);
  Application.Initialize;
```

With this code available on the server, the main form of the client application (which has no connection, as the connection is configured in the secondary forms) has a RadioGroup with the following values:

```
object rgLifeCycle: TRadioGroup
  ItemIndex = 0
  Items.Strings = (
    'Session'
    'Invocation'
    'Server')
end
```

When clicking on a button, the client program reads the current value and passes it as parameter to the server (notice you cannot run the server twice, as you cannot have the same listening socket at the same port opened by two applications at the same time on a computer):

```
procedure TFormDsmcMain.btnStartServerClick(
  Sender: TObject);
var
  aStr: AnsiString;
begin
  Log (rgLifeCycle.Items[rgLifeCycle.ItemIndex]);
  aStr := 'DsnapMethodsServer.exe ' +
    rgLifeCycle.Items[rgLifeCycle.ItemIndex];
  WinExec (PAnsiChar (aStr), CmdShow);
end;
```

Now the main form of the client application also has a button used to create instances of the secondary form, which are destroyed when they are closed (in their `OnClose` event handler), closing the specific connection to the server. Another button is used to log the status of the current client forms:

```
procedure TFormDsmcMain.btnUpdateStatusClick(
  Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to Screen.FormCount - 1 do
    if Screen.Forms[I].ClassType = TFormDsmcClient then
      Log (IntToStr (I) + ': ' +
        Screen.Forms[I].ToString);
end;
```

When calling `ToString` for one of the secondary forms, this returns the status of the connected server object, calling its public `ToString` method:

```
function TFormDsmcClient.ToString: string;
begin
  InitStorageServer;
  Result := StorageServer.ToString;
end;
```

As a first execution example, I've created the server with the default *Session* life cycle, opened two client forms, set the values to 3 and 4, and asked for the overall status, with this result:

```
Session
1: Value: 3 - Object: 1C38400
2: Value: 4 - Object: 1C384E0
```

In a second execution, I've gone for the *Invocation* life cycle, and asking for the overall status twice I saw the following output:

```
Invocation
1: Value: 0 - Object: 1D185B0
2: Value: 0 - Object: 1D18490
1: Value: 0 - Object: 1D185C0
2: Value: 0 - Object: 1D185D0
```

Notice that you are getting a new object for each execution and the objects status is always zero (and any setting will immediately be lost when the object is destroyed immediately after each invocation). Needless to say, this makes sense only for stateless operations.

Finally, I've repeated the same steps (setting values to 3 and 4) with the *Server* life cycle setting, and this time every client form uses the same server object, with the last value set:

```
Server
1: Value: 4 - Object: 1E08490
2: Value: 4 - Object: 1E08490
```

In other words, the practice shows... that the theory is correct! While exploring life cycle configuration in the demo, we've also looked at an example of a client starting the (local) server it needs and of a client with multiple concurrent connections to the server.

PORTING AN OLD DATASNAP DEMO

Having explored some of the alternatives with using DataSnap 2009, let me get back to the most classic usage scenario, which is a multi-tier database application. We've already seen the steps for creating a brand new DataSnap database application. Now let's focus on an equally relevant issue: porting an existing DataSnap (or MIDAS) application to this new architecture.

As a practical example, I've decided to port the ThinPlus application of Mastering Delphi 2005, which showcases a few capabilities of DataSnap, and lets me cover a more complete example, The example also focuses on what needs to be done to port a COM server invoked from a client using a socket to a pure socket-based architecture. The new example (with server and client projects) is in the ThinPlus2009 folder.

Notice that porting DataSnap applications to the new architecture is an interesting option, but not a compulsory one. Traditional DataSnap servers and clients can still compile and work properly in Delphi 2009.

(The program is described in detail in the book Mastering Delphi 2005, but also in previous editions like Mastering Delphi 7. Here I'll provide only an overview of some of its features. Those books can certainly give you a broader picture of the original features of DataSnap and previously MIDAS, which are mostly still available in the Delphi 2009 version.)

PORTING THE SERVER

For porting the server project, I followed these steps:

- I removed the initialization section of the remote data module unit, called AppsRDM. The code removed was the call to the constructor of the `TComponentFactory` class.
- I also removed the `UpdateRegistry` class method of the `TAppServerPlus` class from the same remote data module unit.
- At that point I could eliminate from the uses clause of the remote data module the COM and ActiveX related units: ComServ, ComObj, VCLCom, and StdVcl.
- Next I had to remove the reference to the custom `IAppServerPlus` interface that was used by the project to provide custom server methods (the interface was defined in the project type library).
- I deleted the type library and RIDL file (just created when the project was opened in Delphi 2009) from the project and the disk. I also had to remove a uses statement referring to the type library unit.
- I moved the only server method (`Login`) from the protected section to the public section of the remote data module class, removing from it the `safecall` modifier. As the `TRemoteDataModule` class is already compiled with `$MethodInfo` turned on, there is no need to add this declaration to the project unit.
- Finally, I added to the main form of the program the usual trio of components (server, server class, and server transport), wired them together, and returned the `TAppServerPlus` in the `OnGetClass` event handler of the server class component.

That was all it took to upgrade an old DataSnap server to the Delphi 2009 version. It might seem a lot, but it was actually quite fast. Now it was time to look into the client application, one that does a few custom operations.

UPGRADING THE CLIENT

Porting the client application to DataSnap 2009 is generally easier than porting the server. The core step is to remove the connection components (my demo had three, as it let users experiment with the various connectivity options) and replace it with an SQLConnection and a DSProviderConnection, and make the ClientDataSet component refer to this new remote connection component.

The only specific code I had to change was the call to the `Login` server method. This took place in the `OnAfterConnection` of the connection component, and I've now moved it to the corresponding event of the SQLConnection component:

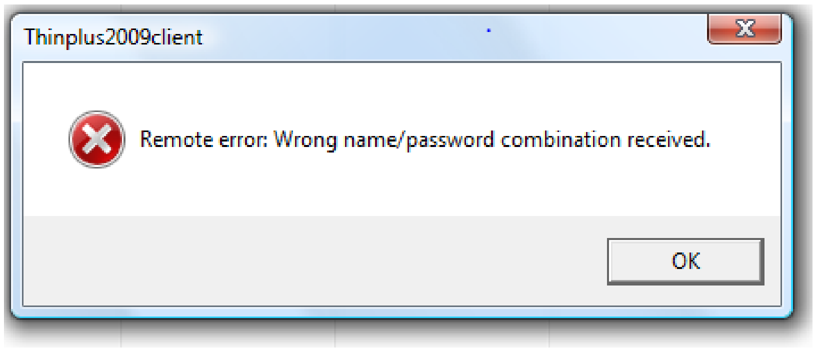
```
procedure TClientForm.SQLConnection1AfterConnect(
  Sender: TObject);
begin
  // was: ConnectionBroker1.AppServer.
  //        Login (Edit2.Text, Edit3.Text);
  SqlServerMethod1.ParamByName('Name').AsString :=
    Edit2.Text;
  SqlServerMethod1.ParamByName('Password').AsString :=
    Edit3.Text;
  SqlServerMethod1.ExecuteMethod;
end;
```

What this call does is to pass client login information to the server. The server validates the information and, only if it succeeds, it will let the provider expose its data. The password check is trivial, but the approach could be interesting. This is the `Login` method on the server:

```
procedure TAppServerPlus.Login(
  const Name, Password: WideString);
begin
  if Password <> Name then
    raise Exception.Create (
      'Wrong name/password combination received');
  ProviderDepartments.Exported := True;
  ServerForm.Add ('Login:' + Name + '/' + Password);
end;
```

Notice that in case the server returns an exception this will be clearly displayed (indicating where it comes from, *Remote error*) on the client side, shown in Figure 3:

Figure 3 Clear Error Messaging for Remote Errors


Thinplus2009client

Remote error: Wrong name/password combination received.

ADVANCED FEATURES OF THINPLUS2009

I upgraded the ThinPlus client and server applications to DataSnap 2009 following the steps mentioned earlier, even if these are some rather complex DataSnap programs, with several customizations. These include fetching data packets manually, using a master/details structure, executing a parametric query, transferring extra data along with the data packets, and the custom remote login I've just covered.

It is worth having a look at these features, even if briefly, as they should help those of you that have never used DataSnap (or not a lot) to appreciate its power. Those who have used it already will figure out how easily the code can be ported to the new architecture. The server application defined a master/details structure, based on the following settings of the (respectively) provider, the master data set, the data source used to refer to it, and the details dataset that refers to the data source:

```
object ProviderDepartments: TDataSetProvider
  DataSet = SQLDepartments
end
object SQLDepartments: TSQLDataSet
  CommandText = 'select * from DEPARTMENT'
  SQLConnection = SQLConnection1
end
object DataSourceDept: TDataSource
  DataSet = SQLDepartments
end
object SQLEmployees: TSQLDataSet
  CommandText =
    'select * from EMPLOYEE where dept_no = :dept_no'
  DataSource = DataSourceDept
  Params = <
    item
      Name = 'dept_no'
      ParamType = ptInput
    end>
  SQLConnection = SQLConnection1
end
```

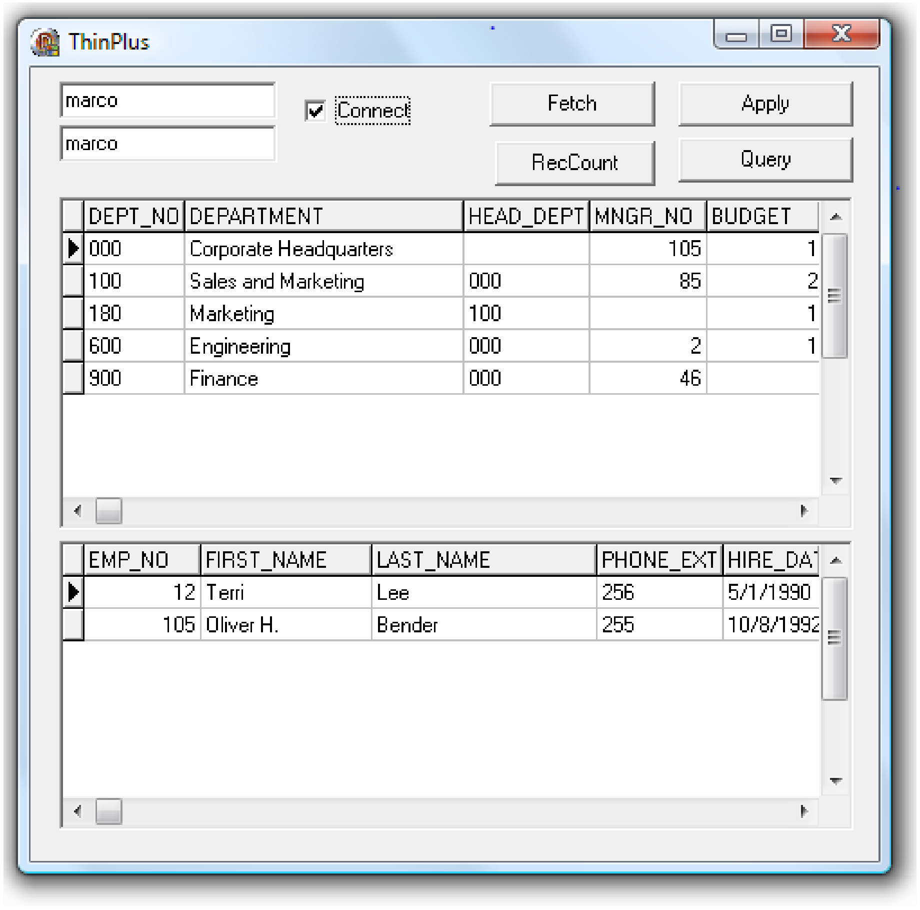
On the client side, the program uses a first ClientDataSet connected with the provider and a second ClientDataSet that refers to a special *data set field* of the first one:

```
object cds: TClientDataSet
  FetchOnDemand = False
  PacketRecords = 5
  ProviderName = 'ProviderDepartments'
  RemoteServer = DSProviderConnection1
  object cdsDEPT_NO: TStringField...
  object cdsDEPARTMENT: TStringField...
  ...
  object cdsSQLEmployees: TDataSetField
    FieldName = 'SQLEmployees'
  end
end
object cdsDet: TClientDataSet
```

```
  DataSetField = cdsSQLEmployees
end
```

The data of the two ClientDataSet components is displayed in two DBGrid controls. Notice how the program fetches only 5 records (as indicated in the `PacketRecords` property) in each data packet, and will stop fetching data after the first packet (as the `FetchOnDemand` property is False), even if the grid in not full. You can see this in the following snapshot of the client user interface just after opening the connection, shown in Figure 4:

Figure 4 Client Interface of the ThinPlus Application



DEPT_NO	DEPARTMENT	HEAD_DEPT	MNGR_NO	BUDGET
000	Corporate Headquarters		105	1
100	Sales and Marketing	000	85	2
180	Marketing	100		1
600	Engineering	000	2	1
900	Finance	000	46	

EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	HIRE_DA
12	Terri	Lee	256	5/1/1990
105	Oliver H.	Bender	255	10/8/1992

Following data packets are fetched manually, as the user clicks the corresponding button:

```
procedure TClientForm.btnFetchClick(Sender: TObject);
begin
  btnFetch.Caption := IntToStr (cds.GetNextPacket);
end;
```

In the button caption, the program shows how many records it fetched in each packet. This will be 5 while there are enough records, then the number or remaining records, and finally zero when all the records have already been retrieved. At each fetch request the client DBGrid will show more data, and its scrollbar will be updated accordingly. You can also use the `bntRecCount` button to ask how many records have been retrieved so far.

The client program has a second form, displayed by pressing the Query button, with another client dataset. This ClientDataSet component is connected with a parametric query defined by the server as:

```
object SQLWithParams: TSQLDataSet
  CommandText =
    'select * from employee where job_code = :job_code'
  Params = <
    item
      DataType = ftString
      Name = 'job_code'
      ParamType = ptInput
      Value = 'Eng'
    end>
  SQLConnection = SQLConnection1
end
```

The client program has a list box, filled at design time with the department names, which is used to pass the proper parameter to the server. Notice that to write this code you have first to update the definition of the parameters, an operation you can do at design time by using the corresponding component editor command for the ClientDataSet component. This is the call used on the client to execute the remote parametric query:

```
procedure TFormQuery.btnParamClick(Sender: TObject);
begin
  cdsQuery.Close;
  cdsQuery.Params[0].AsString := ComboBox1.Text;
  cdsQuery.Open;
  ...
```

On the server, when this query is executed the `OnGetDataSetProperties` event of the provider adds extra information to the returned data packet:

```
procedure TAppServerPlus.
  ProviderQueryGetDataSetProperties(Sender: TObject;
  DataSet: TDataSet; out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['Time', Now, True]);
  Properties[1] := VarArrayOf([
    'Param', SQLWithParams.Params[0].AsString, False]);
end;
```

Notice that the use of variant array parameters still works, even if the transport mechanism used by DataSnap 2009 is now different. On the client side, the `btnParamClick` event handler has two more lines of code to retrieve these extra properties from the data packet:

```
  Caption := 'Data sent at ' + TimeToStr (
    TDateTime (cdsQuery.GetOptionalParam('Time')));
  Label1.Caption := 'Param ' +
    cdsQuery.GetOptionalParam('Param');
```

There are a few more features in DataSnap that have been moved over to the new version, but this overview of the ThinPlus2009 program (mostly unchanged from its original version written in Delphi 6) should be enough for my goals: Show you the power of DataSnap and how easy it is to migrate even a complex application to the Delphi 2009 socket-based (and COM-free) version of DataSnap.

Conclusion

In this paper I've covered one the most significant updates in terms of the component library in Delphi 2009: the new DataSnap architecture for building multi-tier applications without having to resort to COM. You can use DataSnap 2009 for database programming, but also to easily call any server-side method.

Multitier applications based on sockets and with no need for COM registration either at the client or the server-side simplify deployment and make it easier to work with firewalls. DataSnap 2009 leverages the existing DataSnap architecture, a very powerful multi-tier architecture that allows you deploy some of the business logic in the middle tier, opening it up to a more modern (and native) approach. The new DataSnap in Delphi 2009 offers foundations for future extensions, like an HTTP transport protocol.

Finally, the recently shipping Delphi Prim (and new Delphi for .NET development environment by CodeGear) has the ability to create client applications for DataSnap 2009. As an example, you could create an ASP.NET project that connects to a database through a DataSnap 2009 server written in Delphi and compiled for Win32.

It is likely that further extensions will let you use DataSnap for moving data and issuing requests in other scenarios, but the current TCP/IP architecture is already a solid foundation that makes DataSnap 2009 a more flexible solution that its previous version and of many competing approaches.

About The Author

This white paper has been written for Embarcadero Technologies by Marco Cantù, author of the best-selling series Mastering Delphi. The content has been extracted from his latest book "*Delphi 2009 Handbook*", http://www.marcocantu.com/dh2009. You can read about Marco on his blog (http://blog.marcocantu.com) and reach him at his email address: marco.cantu@gmail.com.