White Paper

Delphi 2010 DataSnap: Your data – where you want it, how you want it.

Bob Swart – Bob Swart Training & Consultancy (eBob42)

October 2009

Corporate Headquarters
100 California Street, 12th Floor
San Francisco, California 94111

EMEA Headquarters
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Asia-Pacific Headquarters
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

In this white paper, I will cover the new features and capabilities of Delphi 2010's DataSnap architecture.

1. DataSnap History

Starting in Delphi 3 as MIDAS, with MIDAS II in Delphi 4 and MIDAS III in Delphi 5 when it was a powerful way to build COM-based remote data modules with TCP/IP, HTTP and

(D)COM connection capabilities. Delphi 6 introduced the name DataSnap, and until Delphi 2007 this framework was largely left intact.
Delphi 2009 introduced a re-architecture of DataSnap – removing the dependencies on COM, introducing a more light-weight way to produce remote server objects and client connectivity Initially with only TCP/IP connectivity, but with the ability to build .NET clients using Delphi Prism 2009.
Delphi 2010 is built on top of the DataSnap 2009 architecture and expands this framework with new functionality, including support for new targets using two wizards (VCL Forms, Windows Service, Console but also web targets as ISAPI, CGI or Web App Debugger), HTTP(S) transport protocols, HTTP authentication, client callback functions, support for REST and JSON, and filters to support compression (already built-in) and encryption.

1.1 DataSnap Example Data – *WHERE YOU WANT IT*

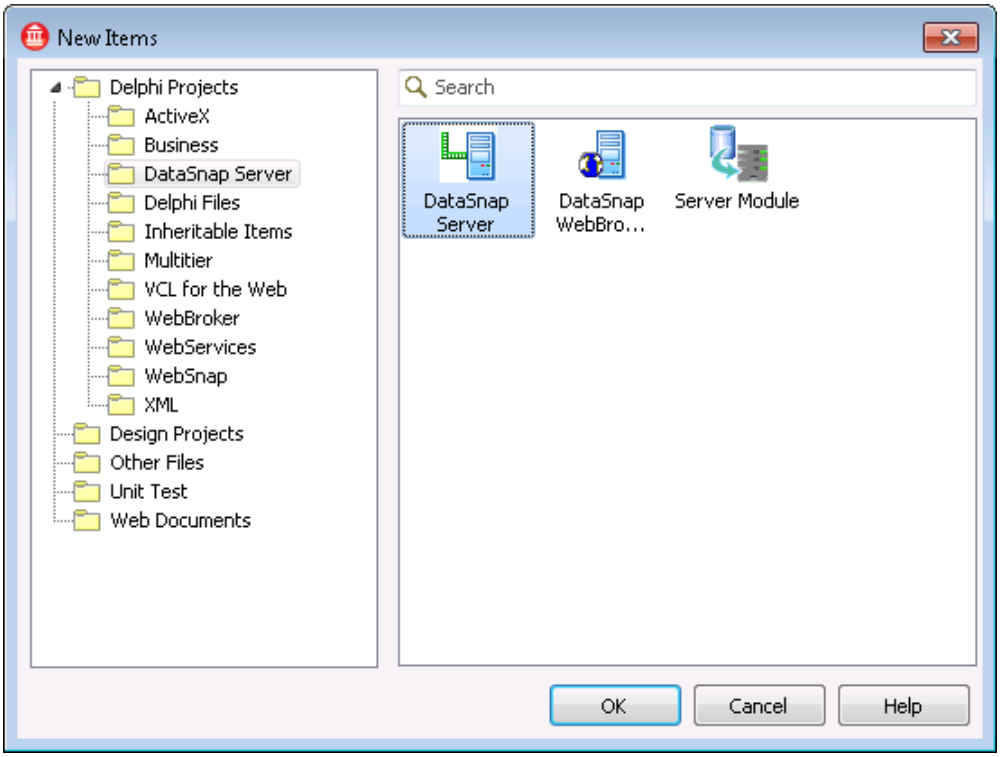
In this white paper, I urge you to play along with the demos and examples. Although Delphi supports many different database systems using DBX4, dbGo for ADO or other data access technologies, in order to make it easy to play along with the examples, I will use DBX4 with BlackfishSQL as DBMS with the employee.jds database which can be found in the directory C:\Documents and Settings\All Users\Documents\RAD Studio\7.0\Demos\database\databases\BlackfishSQL on Windows XP, or C:\Users\Public\Documents\ RAD Studio\7.0\Demos\database\databases\BlackfishSQL on Windows Vista or Windows 7. As you will see from the screenshots, I'm using Windows 7 Professional as operating system for the examples, plus Windows Server 2008 Web Edition for the deployment of the DataSnap ISAPI servers.

2. DataSnap Windows Targets – HOW YOU WANT IT

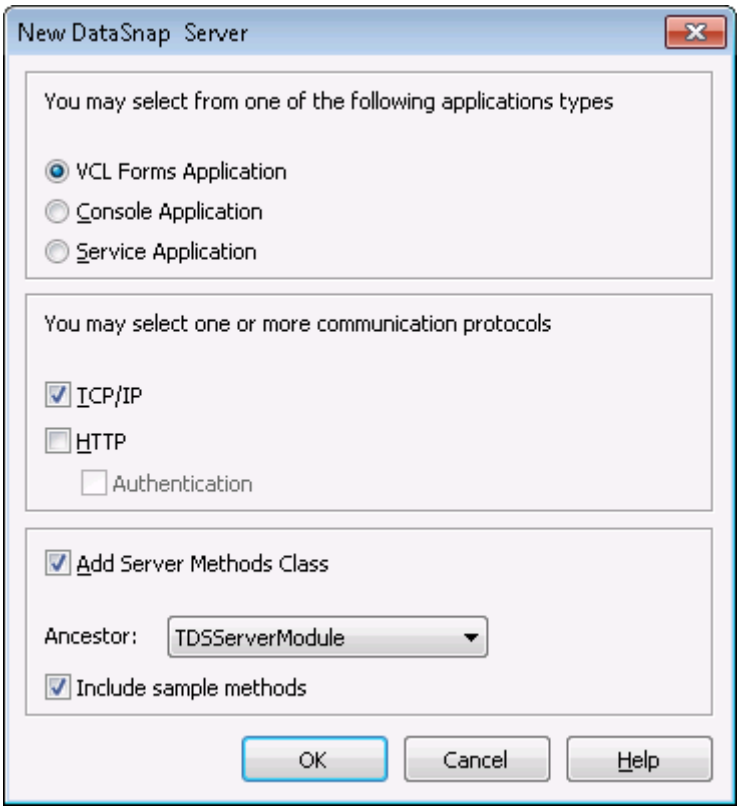
DataSnap 2010 supports three different Windows targets: VCL Forms, Windows Service and Console applications. In this section, I'll discuss the benefits, difference and best cases to use each of these target types.
A sample DataSnap server and client will be built, and we'll cover the TDSServer, TDSServerClass, TDSTCPServerTransport, TDSHTTPService, TDSHTTPWebDispatcher and TDSHTTPServiceAuthenticationManager components as well as the custom server methods and the TDSServerModule class.
Different transport protocols (TCP, HTTP) will be discusses along with their effect and potential benefits. The different options for the lifetime of the DataSnap server object are discussed (server, session and invocation), together with their effect and real-world recommendations. And finally, some deployment issues are covered.

2.1. DataSnap Server Example

There are two different DataSnap Server Wizards in the Object Repository: one to produce Windows based DataSnap Server projects, and one to produce WebBroker based DataSnap Server projects that need to be hosted by a web server like IIS (Internet Information Services). Right now, we'll start with the former.
If you start Delphi 2010, you can find the DataSnap Server wizards in the Object Repository after you do *File* | *New* – *Other*. The DataSnap Server category of the Object Repository shows three icons: DataSnap Server, DataSnap WebBroker Server, and Server Module.



Double-click on the first one (we'll cover the other two later in this white paper), which will produce the following dialog for you:



The first section on this dialog is used to control the target of the project. By default, you will produce a visual VCL Forms application, with a main form. The second choice is the Console application, which gives a console window – probably ideal for tracing what's going on with requests and responses (you can use simple writeln statements to show "what's going on" inside the server application). Both application types are ideal for demos and initial

development, but may be less ideal when you want to deploy the DataSnap server application in the end. Since the new DataSnap architecture is no longer based on COM, an incoming client connection will not be able to "launch" a DataSnap server application. So in order to be able to handle incoming client requests, the DataSnap server should be "up and running" already. And if you want to be able to handle incoming requests on a 24x7 basis, the DataSnap server application should be up and running during that time as well. For a VCL Forms or Console application, this means that an account must be logged on to Windows, running the DataSnap server application, which is far from ideal. The third choice is actually better in that case: a Windows Service application, which can be installed and configured to run automatically when the machine is started (without the requirement that someone has to be logged into the machine). The downside of a service application is that by default it doesn't show itself on the desktop, and it's also a bit harder to debug. However, in order to be able to get the best out of all three worlds, I will show you in a minute how to create a project group for a VCL Forms DataSnap server application, a Console DataSnap server application as well as a Windows Service DataSnap server application, all sharing the same custom server methods, thereby allowing you a single DataSnap server application that can be compiled to (and deployed as) three different targets when needed.

The second section of the New DataSnap Server dialog shows the different communication protocols that we can use. Compared to DataSnap 2009, we can now also have HTTP communication, as well as HTTP Authentication. In order to be most flexible, I suggest we check all options here, so we can use both TCP/IP and HTTP, and use HTTP Authentication in combination with HTTP as well.

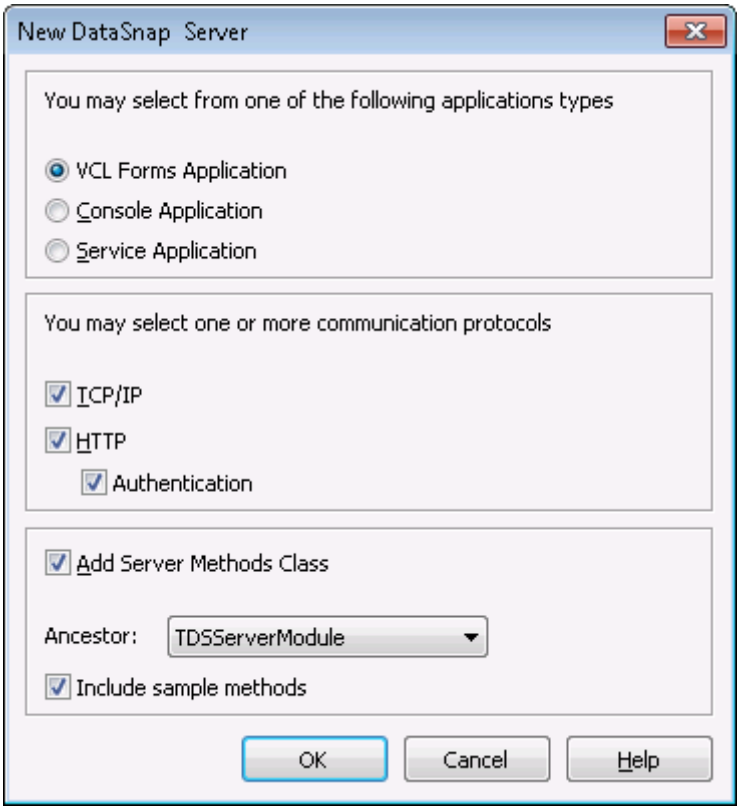
The final section of the New DataSnap Server dialog is already configured fine in my view. If offers us the benefit of generating a server method class, and we can even pick the ancestor type: TPersistent, TDataModule or TDSServerModule. The last choice is the best, enabling RTTI for methods right from the start (although there may be circumstances where you feel a regular TDataModule or perhaps – if you don't use any datasets or non-visual controls - even a TPersistent is enough).
A little snippet from unit DSServer.pas shows the relation between TDSServerModule and the TProviderDataModule, which in turn is derived from TDataModule.

```
TDSServerModuleBase = class(TProviderDataModule)
public
  procedure BeforeDestruction; override;
  destructor Destroy; override;
end;

{$MethodInfo ON}
TDSServerModule = class(TDSServerModuleBase)
end;
{$MethodInfo OFF}
```

When unsure what to do, select the TDSServerModule as ancestor class.

2.1.1. MULTI-TARGET PROJECT GROUP – VCL FORMS

As promised, let's create a multi-target DataSnap Server project group now. First, let's start with the VCL Forms Application as DataSnap Server, with all communication protocols selected.



The result is a new project – by default called Project1.dproj - with three units, by default called ServerContainerUnit1.pas, ServerMethodsUnit1.pas and Unit1.pas. We should first do File | Save Project As, specifying a bit more detailed filenames in some cases. Save Unit1.pas in file MainForm.pas, save ServerContainerUnit1.pas in ServerContainerUnitDemo.pas, save ServerMethodsUnit1.pas in ServerMethodsUnitDemo.pas and save Project1.dproj in DataSnapServer.dproj.

We will add the Console Application and Service Application to the project group in a minute. First, let's examine what we have at this point, and try to compile the project. If you compile the DataSnapServer project, it should give you one error message (which is my fault, since we renamed the generated unit ServerMethodsUnit1.pas): The error message is caused by the ServerContainerUnitDemo.pas unit that contains the ServerMethodsUnit1 unit in the uses clause of its implementation section (line 30), while I saved ServerMethodsUnit1.pas as ServerMethodsUnitDemo. In order to fix this issue, change the uses clause to reflect this unit rename as well, and recompile again. This time, an error will occur on line 37 where the ServerMethodsUnit1 is used as qualifier for the TServerMethods1 class. Change ServerMethodsUnit1 to ServerMethodsUnitDemo here as well, so we can compile the DataSnap Server project without any further issues now.

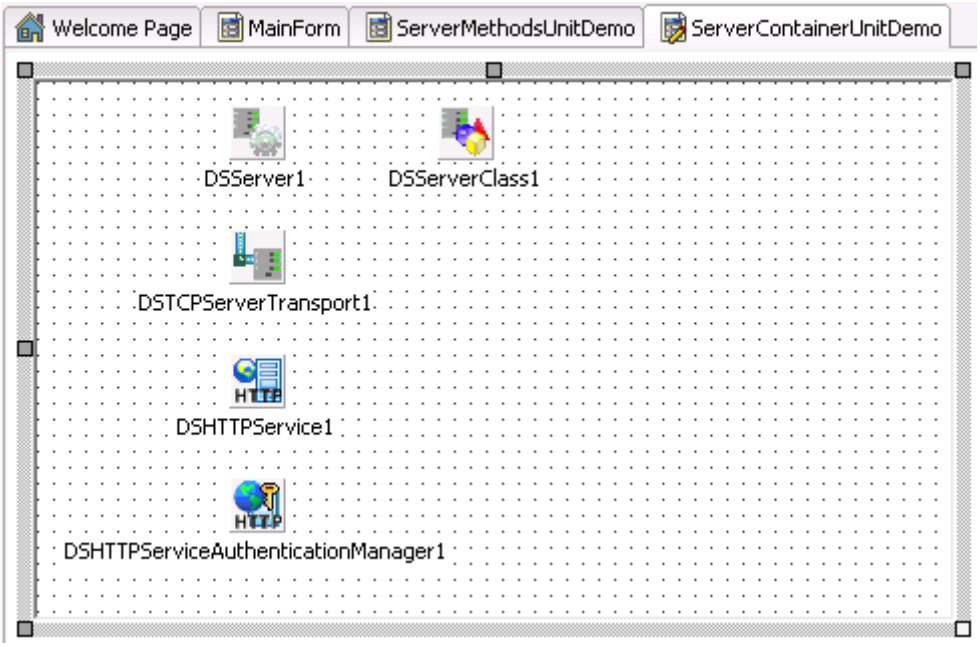
The implementation section of unit ServerContainerUnitDemo should now look as follows:

```
implementation
uses
  Windows, ServerMethodsUnitDemo;

{$R *.dfm}

procedure TServerContainer1.DSServerClass1GetClass(
  DSServerClass: TDSServerClass; var PersistentClass: TPersistentClass);
begin
  PersistentClass := ServerMethodsUnitDemo.TServerMethods1;
end;
```

```
end.
```

2.1.1.1. SERVERCONTAINERUNITDEMO

If you take a look at the Design tab of the ServerContainerUnitDemo unit, you'll see no less than five components: a TDSServer, a TDSServerClass, a TDSTCPServerTransport (for the TCP/IP communication), a TDSHTTPService (for the HTTP communication) and a TDSHTTPServiceAuthenticationManager component (for the HTTP Authentication).



The first two are always included, the other three are included based on the selected communication protocol options of course.

2.1.1.1.1. TDSSERVER

The TDSServer component has only four properties: AutoStart, HideDSAdmin, Name and Tag. The AutoStart property is set to True by default, which means that the DataSnap Server will start as soon as the form is created. If you don't set AutoStart to True, you can manually call the Start method, and you can always call the Stop method. You can use the Started function to find out if the DataSnap Server is already started.
The HideDSAdmin property is set to False by default. If you set it to True, then clients connecting to this DataSnap Server cannot call the built-in server methods from the TDSAdmin class. The TDSAdmin class is not actually a class, but the TDSAdmin methods that we can call are documented in the DSNames unit:

```
TDSAdminMethods = class
public
  const CreateServerClasses = 'DSAdmin.CreateServerClasses';
  const CreateServerMethods = 'DSAdmin.CreateServerMethods';
  const FindClasses = 'DSAdmin.FindClasses';
  const FindMethods = 'DSAdmin.FindMethods';
  const FindPackages = 'DSAdmin.FindPackages';
  const GetPlatformName = 'DSAdmin.GetPlatformName';
  const GetServerClasses = 'DSAdmin.GetServerClasses';
  const GetServerMethods = 'DSAdmin.GetServerMethods';
  const GetServerMethodParameters = 'DSAdmin.GetServerMethodParameters';
  const DropServerClasses = 'DSAdmin.DropServerClasses';
  const DropServerMethods = 'DSAdmin.DropServerMethods';
  const GetDatabaseConnectionProperties = 'DSAdmin.GetDatabaseConnectionProperties';
```

```
end;
```

The TDSServer component has five events: OnConnect, OnDisconnect, OnError, OnPrepare and OnTrace. We can write event handlers for these five events to respond to the different situations, for example by writing a line of text to a log file.
The OnConnect, OnDisconnect, OnError and OnPrepare events have an argument derived from TDSEventObject, which contains properties for the DxContext, the Transport, the Server and the DbxConnection component. The TDSConnectEventObject type, used for the OnConnect and OnDisconnect also contains ConnectionProperties as well as ChannelInfo. The TDSErrorEventObject also includes the Exception that caused the error, and the TDSPrepareEventObject includes properties for the MethodAlias and the ServerClass that we want to use.
The OnTrace event handler has an argument of type TDBXTraceInfo. Note that this generated OnTrace event handler will also generate some code insight errors, as the types TDBXTraceInfo and CBRType are unknown to the compiler. In order to solve that, we need to add the DBXCommon unit (for the TDBXTraceInfo type) and the DBCommonTypes unit (for CBRType).
During the OnConnect event handler, we can examine the ChannelInfo for the connection, for example (using a custom function LogInfo to write this information to a logfile):

```
procedure TServerContainer1.DSServer1Connect(
  DSConnectEventObject: TDSConnectEventObject);
begin
  LogInfo('Connect ' + DSConnectEventObject.ChannelInfo.Info);
end;
```

And inside the OnTrace event handler we can log the contents of the TraceInfo.Message to get a good idea what the server is doing.

```
function TServerContainer1.DSServer1Trace(TraceInfo: TDBXTraceInfo): CBRType;
begin
  LogInfo('Trace ' + TraceInfo.CustomCategory);
  LogInfo('  ' + TraceInfo.Message);
  Result := cbrUSEDEF; // take default action
end;
```

Note that at the client side you can also trace the communication between the DataSnap client and server by using a TSQLMonitor component connected to the TSQLConnection component (something which I'll demonstrate when we're building the client for this DataSnap server).

An example of the trace output can be as follows:

```
17:05:55.492 Trace
17:05:55.496   read 136 bytes:{"method":"reader_close","params":[1,0]}
               {"method":"prepare","params":[-1,false,"DataSnap.ServerMethod",
                "TServerMethods1.AS_GetRecords"]}
17:05:55.499 Prepare
```

As you can see, the TraceInfo.Message contains information about the number of bytes as well as the method that was called.

2.1.1.1.2. TDSSERVERCLASS

The TDSServerClass component is responsible for specifying the class at the server side which is used to expose published methods to the remote clients (using dynamic method invocation).
The TDSServerClass component has a Server property that points to the TDSServer component. The other important property – apart from the Name and Tag properties – is the

LifeCycle property. By default it's set to Session, but it can also be set to Server or Invocation. From long to short, the meaning of Server, Session and Invocation mean that one instance of the class is used for the entire lifetime of the server, for the lifetime of the DataSnap session, or for the single invocation of a method. Session means that each incoming connection will get its own instance of the server class. If you change this to Invocation, you will end up with a state-less server class – for example useful if you want to deploy a CGI web server application (which is also stateless, and loaded/unloaded for every request). When you change the LifeCycle to Server, it means that a single server class instance is shared with all incoming connections and requests. This can be useful if you want to "count" the number of requests for example, but you must ensure that no threading issues can occur (when multiple requests come in and expect to be handled simultaneously).

The TDSServerClass component has four events: OnCreateInstance, OnDestroyInstance (these are fired when the instance is created or destroyed obviously), OnGetClass and OnPrepare. The OnPrepare event handler can be used to prepare a server method.
When using Delphi 2009, or when using Delphi 2010 but manually placing the TDSServerClass component on a container, the OnGetClass event needs to be implemented by us, since we need to specify which class will be "remoted" from the server to the client. The DataSnap wizards in Delphi 2010, however, will now automatically implement the OnGetClass event handler for us, as follows:

```
procedure TServerContainer1.DSServerClass1GetClass(
  DSServerClass: TDSServerClass; var PersistentClass: TPersistentClass);
begin
  PersistentClass := ServerMethodsUnitDemo.TServerMethods1;
end;
```

Note that this was the generated code we had to modify slightly when we renamed the generated unit ServerMethodsUnit1 and saved it in ServerMethodsUnitDemo.pas.

2.1.1.1.3. TDSTCPSERVERTRANSPORT

The TDSTCPServerTransport components is responsible for the communication between the DataSnap server and the DataSnap clients, and is using TCP/IP as communication protocol.

The TDSTCPserverTransport component has five important properties: BufferKBSize, Filters (new in Delphi 2010), MaxThreads, PoolSize, Port, and Server.
The BufferKBSize property specifies the size of the buffer for the communication, and is set to 32 (KB) by default. The Filters property can contain a collection of transport filters, which will be covered in detail in section 4. The MaxThreads property can be used to define a maximum number of threads (by default set to 0 for no maximum or limit). The PoolSize can be used to enable connection pooling, and the Port property can be used to control the TCP/IP port that the server uses to connect to the client (if you change that here, you also need change it at the DataSnap client later).
The Server property is pointing to the TDSServer component. The DSTCPServerTransport component has no events.

2.1.1.1.4. TDSHTTPSERVICE

The TDSHTTPService component is responsible for the communication between the DataSnap server and the DataSnap clients using the HTTP protocol.

The TDSHTTPService component has 10 properties (apart from the Name and Tag property): Active, AuthenticationManager, DSHostname, DSPort, Filters, HttpPort, Name, RESTContext, Server, and the read-only ServerSoftware property.

The Active property specifies if the DSHTTPService is listening to incoming requests. It can be set to True at design-time, but this will prevent the DataSnap Server application from starting at run-time (since there can only be one active DSHTTPService component listening to the same port – and with one already active at design-time, you cannot start another one at run-time). Your best way to start the TDSHTTPService is to set Active to True in the OnCreate event handler of the TServerContainer:

```
procedure TServerContainer1.DataModuleCreate(Sender: TObject);
begin
  DSHTTPService1.Active := True;
end;
```

The AuthenticationManager property is used to define the manager component for handling HTTP authentication, and in our example is already pointing to the TDSHTTPServiceAuthenticationManager component. This component is covered in more detail in section 2.1.1.1.5.
The DSHostname and DSPort properties are used to define the DataSnap server to connect to, but are only used when the Server property is not set. In most cases, you just connect the Server property to a TDSServer component instead.
The Filters property can contain a collection of transport filters, which will be covered in detail in section 4.
The HttpPort property is used to define which port the DSHTTPService component will listen to for incoming connections. Note that by default this port value is set to 80, which is something you must change if you develop on a machine which has a web server (like Internet Information Services IIS) installed, since the web server will already use port 80, which means that the TDSHTTPService component cannot also start to listed to port 80. You'll get an error when trying to run the application, with a message that may not be immediately clear.
The RESTContext property defines the REST context URL that we can use to call the DataSnap server as a REST service. By default, the RESTContext is set to "rest",. which means we can call the server as http://localhost/datasnap/rest/... as I'll demonstrate in section 6 on REST, JSON and client callback methods.
Finally, the Server property should be pointed to a TDSServer component on the same container. If you do not want to connect the Server property, you can use the DSHostname and DSPort properties to connect to the DataSnap Server using TCP. When the Server property is set, the DSHostname and DSPort properties are ignored.

The TDSHTTPService component has five events: four REST related, and one Trace event. The REST events will be covered in more detail in section 6.
The OnTrace event can be used to trace the calls to the DSHTTPService component, for example as follows:

```
procedure TServerContainer1.DSHTTPService1Trace(Sender: TObject;
  AContext: TDSHTTPContext; ARequest: TDSHTTPRequest;
  AResponse: TDSHTTPResponse);
begin
  LogInfo('HTTP Trace ' + AContext.ToString);
  LogInfo('  ' + ARequest.Document);
  LogInfo('  ' + AResponse.ResponseText);
end;
```

Note that the HTTP Trace information will only be shown when the client is actually using HTTP to connect to the server (and not the default TCP/IP protocol instead).

An example of the trace output can be as follows:

```
17:05:55.398 HTTP Trace TDSHTTPContextIndy
17:05:55.400   /datasnap/tunnel
17:05:55.403   OK
```

Which means the AContext is set to TDSHTTPContextIndy, the ARequest is set to /datasnap/tunnel, and the AResponse is set to OK.

2.1.1.1.5. TDSHTTPSERVICEAUTHENTICATIONMANAGER

The TDSHTTPServiceAuthenticationManager component will be placed on the Server Container when the Authentication checkbox for the HTTP communication protocol has been checked. But you can also manually place it on the server container, of course. The TDSHTTPServiceAuthenticationManager component must be connected to the AuthenticationManager property of a TDSHTTPService component.

The TDSHTTPServiceAuthenticationManager component has one event: the OnHTTPAuthenticate event, which can be used to check the HTTP Authentication information which can be supplied from the DataSnap Client to the Server.

```
procedure TServerContainer1.DSHTTPServiceAuthenticationManager1HTTPAuthenticate(
  Sender: TObject; const Protocol, Context, User, Password: string;
  var valid: Boolean);
begin
  if (User = 'Bob') and (Password = 'Swart') then
    valid := True
  else
    valid := False
end;
```

Obviously, you should extend this validation routine with a lookup in a database with a hashed version of the password for example. HTTP Authentication where the User and (hashed) Password information is sent from the client to the server is best done using HTTPS by the way, so I'm hoping that Embarcadero will add a HTTPS protocol to the current list of HTTP and TCP/IP protocols.
HTTPS will ensure that the connection is secure and the data packet is encrypted, so other people with data sniffers will not be able to get their hands on your User and (hashed) Password information. Consult with your ISP or web master to find out if there are HTTPS possibilities for your domain – it's highly recommended (I'm using https://www.bobswart.nl for example).

Another good technique used in some real-world DataSnap Server applications is to write the HTTP Authentication (attempts) to a logfile, where you then also write out the Protocol and Context information. This can give you an idea who is logging in, and who is trying to login (for example fake login attempts by people trying to get access to your DataSnap Server).

2.1.1.2. SERVERMETHODSUNITDEMO

Now that we've examined the ServerContainerUnitDemo.pas unit, it's time to switch to another important unit of the new DataSnap server application: the ServerMethodsUnitDemo.pas unit. In the New DataSnap Server dialog, we specified the TDSServerModule class to use as ancestor, so the TServerMethods1 type is derived from TDSServerModule (which is derived from TDSServerModuleBase, which in turn is derived from TProviderDataModule, adding a Destroy destructor and a BeforeDestruction procedure. A TProviderDataModule is derived from a normal TDataModule, adding the capabilities to work with providers (more about this later).

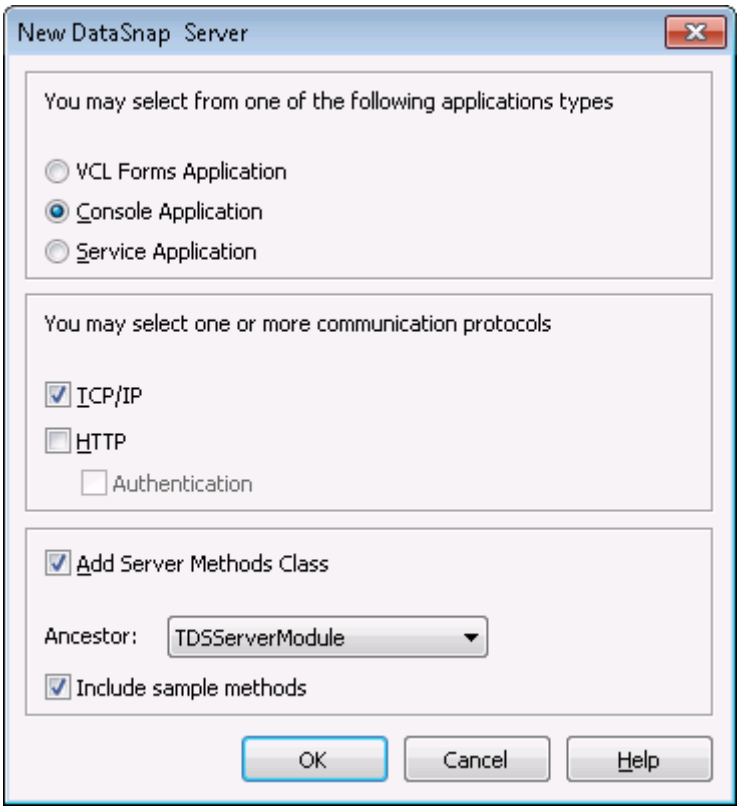
Because one of the ancestor classes of TServerMethods1 is a TDataModule, the design tab will show us the designer area for a data module: we can place any non-visual control such as data-access controls here. We'll place data-access components in section3, but for now we

should leave the design area empty, and just stick with adding methods to the TServerMethods1 class.

If you do not want add additional targets to the project group – for a DataSnap Console application and/or a DataSnap Windows Service application, feel free to skip to section 2.1.4 when we start implementing the Server Methods.

2.1.2. MULTI-TARGET PROJECT GROUP – CONSOLE

Time to return to the project group, and add a second target: this time the Console application. Right-click on the project group node, and select Add New Project. In the Object Repository, again go to the DataSnap Server category and double-click on the DataSnap Server icon. This time, select the Console Application target in the New DataSnap Server dialog.



Note that it doesn't matter what other options you'll select: we're going to reuse the ServerContainerUnitDemo.pas and ServerMethodsUnitDemo.pas from the DataSnapServer project anyway.

Click on OK to generate the new project. It will generate a Project1.dproj again, plus a ServerContainerUnit2.pas and a ServerMethodsUnit2.pas. Right-click on the ServerMethodsUnit2.pas unit in the Project Manager, and remove it from the new project. Keep the ServerContainerUnit2.pas for now, because we need to copy one method from it. Save the project in DataSnapConsoleServer.dproj.
If you compare the contents of the ServerContainerUnitDemo.pas and the newly generated unit ServerContainerUnit2.pas (for the console application), you'll note that the latter includes a global procedure called RunDSServer. This global procedure is only useful for a Console application, so we should copy it and paste it inside the DataSnapConsoleServer.dproj project source code, just before the begin of the main block.

Error insight will then flag a number of issues, which can be resolved by adding the Windows unit to the uses clause of DataSnapConsoleServer.dpr. The DataSnapConsoleServer should now look as follows:

```
program DataSnapConsoleServer;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Windows,
  ServerContainerUnit2 in 'ServerContainerUnit2.pas'
    {ServerContainer2: TDataModule};

procedure RunDSServer;
var
  LModule: TServerContainer2;
  LInputRecord: TInputRecord;
  LEvent: DWord;
  LHandle: THandle;
begin
  Writeln(Format('Starting %s', [TServerContainer2.ClassName]));
  LModule := TServerContainer2.Create(nil);
  try
    LModule.DSServer1.Start;
    try
      Writeln('Press ESC to stop the server');
      LHandle := GetStdHandle(STD_INPUT_HANDLE);
      while True do
      begin
        Win32Check(ReadConsoleInput(LHandle, LInputRecord, 1, LEvent));
        if (LInputRecord.EventType = KEY_EVENT) and
        LInputRecord.Event.KeyEvent.bKeyDown and
        (LInputRecord.Event.KeyEvent.wVirtualKeyCode = VK_ESCAPE) then
          break;
      end;
    finally
      LModule.DSServer1.Stop;
    end;
  finally
    LModule.Free;
  end;
end;


begin
  try
    RunDSServer;
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end
end.
```

Unfortunately, we should now perform three more actions that will make Error Insight show some problems again: the DataSnapConsoleServer project still uses the ServerContainerUnit2.pas, and I want to remove that unit now (so right-click on it, and select Remove From Project. Then, right-click on the DataSnapConsoleServer.exe node, select Add and select the ServerContainerUnitDemo.pas as well as the ServerMethodsUnitDemo.pas units and add them to the project.
As a result, all references to TServerContainer2 should now be flagged as a syntax error. The ServerMethodsUnitDemo.pas should define the type TServerContainer1, so in order to fix the last problems: rename TServerContainer2 in the source code from DataSnapConsoleServer to TServerContainer1 (in three places in total).

Then, we should be able to compile both the new DataSnapConsoleServer.dproj project as well as the original DataSnapServer.dproj project. Sharing the ServerContainerUnitDemo.pas as well as the ServerMethodsUnitDemo.pas units between the two project targets.

2.1.3. MULTI-TARGET PROJECT GROUP – WINDOWS SERVICE

With the DataSnap VCL Forms server and the DataSnap console server projects in the project group, there's one target left to add: the DataSnap Windows Service application. In order to add this target, right-click on the project group node, select Add New Project and from the Object Repository double-click on the New DataSnap Server icon again. This time, select the Service Application and also all communication protocol option (TCP/IP, HTTP as well as HTTP Authentication). As we'll see in a moment, the Server Container for a Service application is slightly different from a Server Container inside a VCL Forms or Console application, so we should be sure to select all communication protocol options here as well. Again the result of the New DataSnap Server dialog is a new project, plus a ServerContainerUnit and a ServerMethodsUnit. The ServerMethodsUnit is the same as we've seen before, so we can remove it from the new project, and replace it with the ServerMethodsUnitDemo.pas we've been sharing with the VCL Forms and Console DataSnap applications.
The new ServerCotainerUnit1.pas unit is different however: instead of using a TDataModule to place the TDSServer, TDSServerClass and the transport components, we now get a class derived from TService containing the DataSnap components. And apart from being derived from TService, there are also four special methods added to it, to implement the Stop, Pause, Continue and Interrogate events of the service:

```
type
  TServerContainer3 = class(TService)
    DSServer1: TDSServer;
    DSTCPServerTransport1: TDSTCPServerTransport;
    DSHTTPService1: TDSHTTPService;
    DSHTTPServiceAuthenticationManager1: TDSHTTPServiceAuthenticationManager;
    DSServerClass1: TDSServerClass;
    procedure DSServerClass1GetClass(DSServerClass: TDSServerClass;
      var PersistentClass: TPersistentClass);
    procedure ServiceStart(Sender: TService; var Started: Boolean);
  private
    { Private declarations }
  protected
    function DoStop: Boolean; override;
    function DoPause: Boolean; override;
    function DoContinue: Boolean; override;
    procedure DoInterrogate; override;
  public
    function GetServiceController: TServiceController; override;
  end;
```

In other words: we cannot share the original ServerContainerUnitDemo.pas unit with the Windows Service project, and we should rename the ServerContainerUnit1.pas to ServerContainerUnitServiceDemo.pas. And while we're at it, let's save the project itself to DataSnapServiceServer.dproj.
We need to fix the reference to the old ServerMethodsUnit2.pas unit in the ServerContainerUnitServiceDemo, and change it to ServerMethodsUnitDemo.pas, so at least we're sharing the same Server Methods unit between all three targets.

2.1.4. SERVER METHODS

When you have one or more DataSnap Server projects, all sharing the same ServerMethodsUnitDemo unit, it's time to examine the server methods in some more detail.

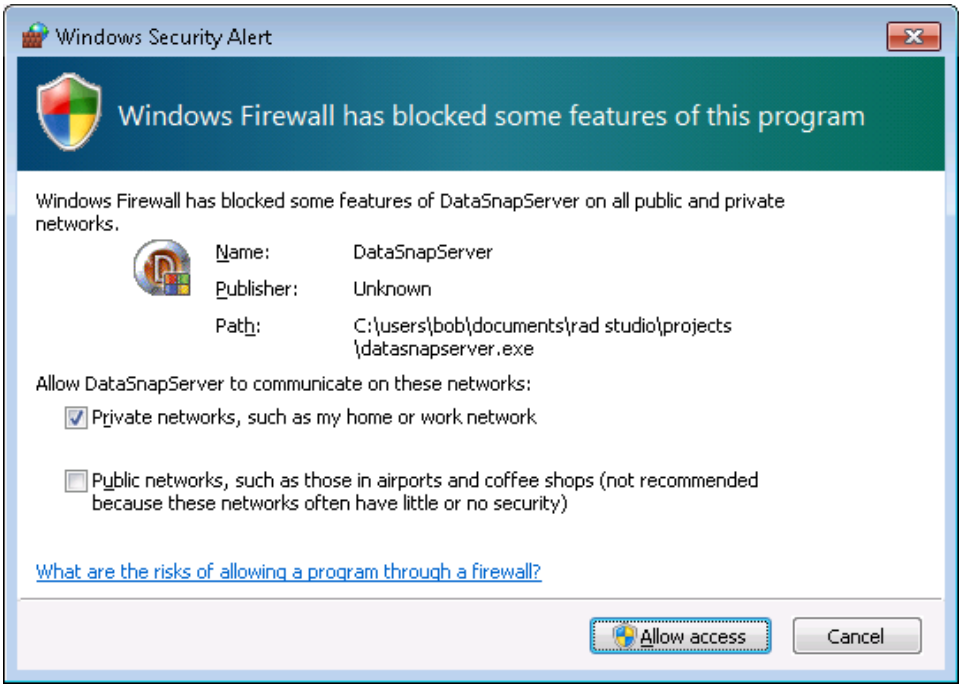
As I mentioned before, the TServerMethod1 class is the DataSnap server object that exposes the methods (using RTTI) to be exposed to the DataSnap clients. If you checked the "Include sample methods" option, there will be one sample method already present in the TServerMethods1 class: function EchoString. Let's add another one, to return the current time from the DataSnap server machine. To do that, modify the definition of TServerMethods1 to include two public methods as follows:

```
type
  TServerMethods1 = class(TDSServerModule)
  private
    { Private declarations }
  public
    { Public declarations }
    function EchoString(Value: string): string;
    function ServerTime: TDateTime;
  end;
```

The implementation of the ServerTime method is really easy, almost as short as the example method EchoString:

```
function TServerMethods1.EchoString(Value: string): string;
begin
  Result := Value;
end;

function TServerMethods1.ServerTime: TDateTime;
begin
  Result := Now
end;
```

We can now compile and Run server application. If you created multiple project targets, then the easiest one to test right now is the DataSnapServer executable. Depending on your version of Windows and level of security settings, you may see a Windows Security Alert to tell you that Windows Firewall has blocked some features of the DataSnapServer application.

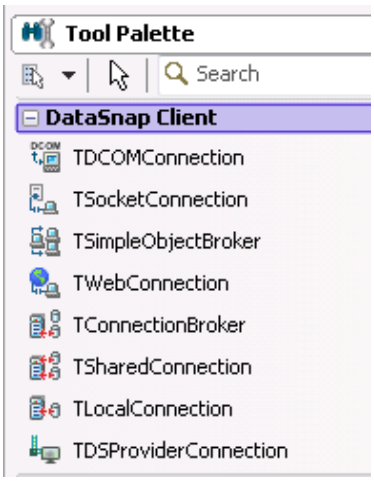


This is due to the fact that the DataSnapServer application is actively listening to incoming requests over TCP/IP and HTTP at this time. Like a Trojan Horse, only one that we want to be able to listen to incoming requests, so click on the button to Allow Access and continue to start the DataSnap Server.

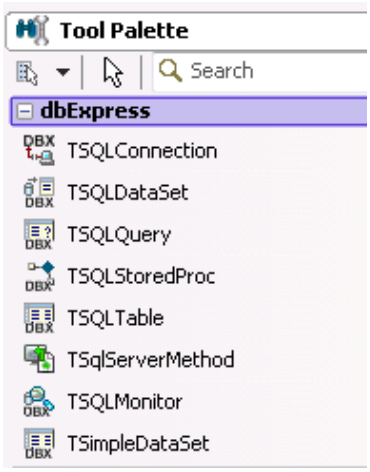
2.2. DATASNAP CLIENT

When the first DataSnap Server demo is up-and-running and listening to incoming requests, it's time to build a client. In this section, I'll explain how we can connect from the client to the server, how to import the methods by generating server classes.

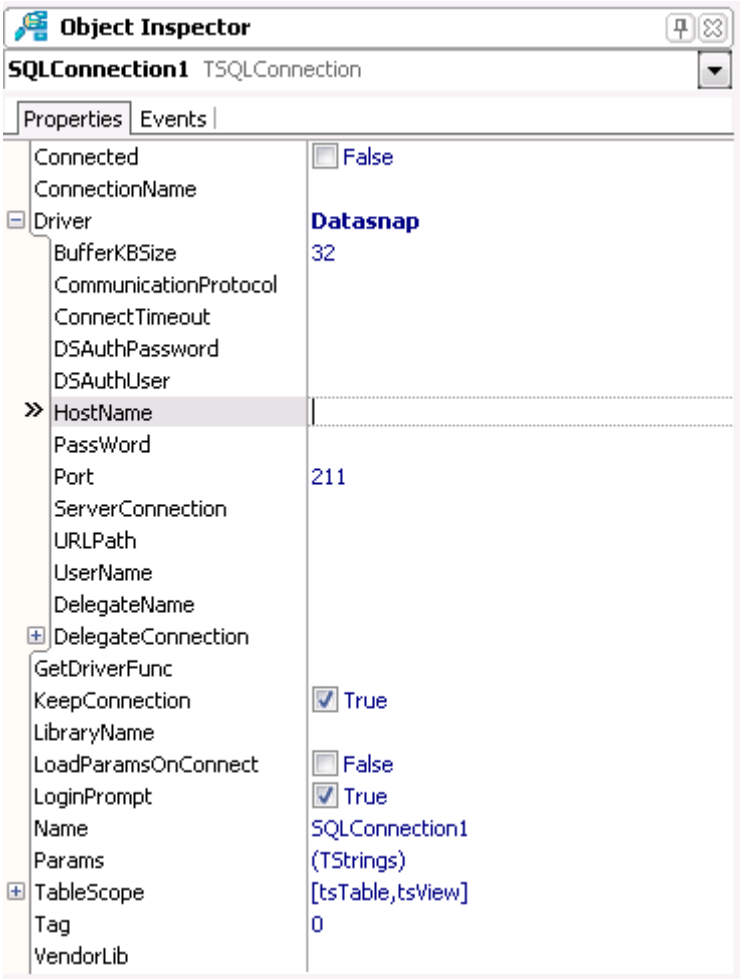
Make sure the DataSnap server is up an running, so we can create the DataSnap client application project. In order to easily switch between the DataSnap Server projects and the DataSnap Client project at design-time, we can add the DataSnap client project to the same project group. Any project can be a DataSnap client, but for this demo I'm using a VCL Forms application and save it in DataSnapClient.dpr with the form in ClientForm.pas. Where the DataSnap Server category of the Tool Palette contains the six DataSnap (server) components, the DataSnap Client category does not contain the components we need to use right now:



The components in the DataSnap Client category are the "old" DataSnap components, which can still be used, but are no longer the recommended way of using DataSnap. There is one exception: the new TDSProviderConnection component, which can be used to connect an "old" DataSnap server to a new DataSnap client (something I'll show in section 3.2).
Instead of the DataSnap Client category, we must look at the dbExpress category, where a new component called TSQLServerMethod can be found (note from the next screenshot that this new component can be identified easily, since it's the one with TSql in its prefix instead of TSQL that the existing dbExpress components still have).



The TSqlServerMethod component can be used to call remote methods from a DataSnap server, but it needs to connect to the DataSnap server first.
The connecting can be done using a TSQLConnection component – and not using one of the older TxxxConnection components. In order to facilitate this, the TSQLConnection component has a new driver name in the Driver drop-down list, called DataSnap.
So, place a TSQLConnection component on the ClientForm, and set its Driver property to DataSnap. As a result, the Driver property will show a plus sign (on the left of the property name), and we can expand the property to show all subproperties.



Note that by default, if the CommunicationProtocol property is left empty, the TSQLConnection will use TCP/IP as protocol (over the specified port 211).
The BufferKBSize is set to 32 (KB), and the Port is set to 211 by default – just as on the Server side. For real-world situations, I always change the Port number from 211 to something else (at both the server side and the client side), since Port 211 is a bit too well-known as the port to connect to DataSnap servers.
The HostName, UserName and Password are used to connect to the DataSnap server. For a local test, we can set the HostName set to localhost, but in general you can enter any host name, DNS name or direct IP number as value for this property.
Do not forget to set the LoginName property of the TSQLConnection component to False to avoid the login dialog.
Once the TSQLConnection Driver properties have been specified, we can set the Connected property to True in order to (try to) make a connection to the DataSnap server. Note that the DataSnap Server must be running at the server side in order to make the connection!

2.2.1. DATASNAP CLIENT CLASSES

Once you've verified that the connection can be made, we can right-click on the TSQLConnection component and select the Generate DataSnap client classes option, This will produce a new unit, by default called Unit1, with a class called TServerMethods1Client (the actual name of the DataSnap Server methods class at the server side with the "Client" part added to it). Save this unit in ServerMethodsClient.pas.
The definition of the generated TServerMethods1Client class should be as follows:

```
type
  TServerMethods1Client = class
  private
    FDBXConnection: TDBXConnection;
    FInstanceOwner: Boolean;
    FEchoStringCommand: TDBXCommand;
    FServerTimeCommand: TDBXCommand;
  public
    constructor Create(ADBXConnection: TDBXConnection); overload;
    constructor Create(ADBXConnection: TDBXConnection;
      AInstanceOwner: Boolean); overload;
    destructor Destroy; override;
    function EchoString(Value: string): string;
    function ServerTime: TDateTime;
  end;
```

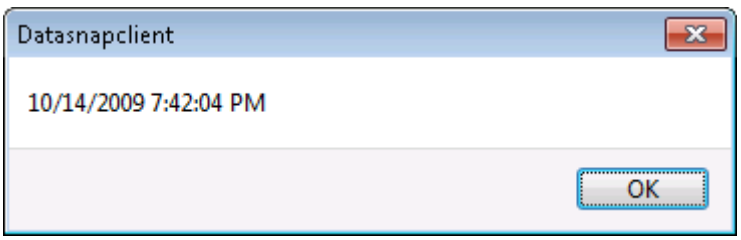
As you can see, there are two constructors for the TServerMethods1Client class, one destructor and there should be the two server methods we defined at the DataSnap server side.

In order to use these methods, add the ServerMethodsClient unit to the uses clause of the ClientForm, place a TButton component on the client form, and write the following code in the OnClick event handler:

```
procedure TForm2.Button1Click(Sender: TObject);
var
  Server: TServerMethods1Client;
begin
  Server := TServerMethods1Client.Create(SQLConnection1.DBXConnection);
  try
    ShowMessage(DateTimeToStr(Server.ServerTime))
  finally
    Server.Free
  end;
end;
```

This code will create the instance of the TServerMethods1Client, then call the ServerTime server method, and finally destroy the proxy to the DataSnap server again.

Clicking on this button will produce the messagebox with the value of the current server time, as expected.

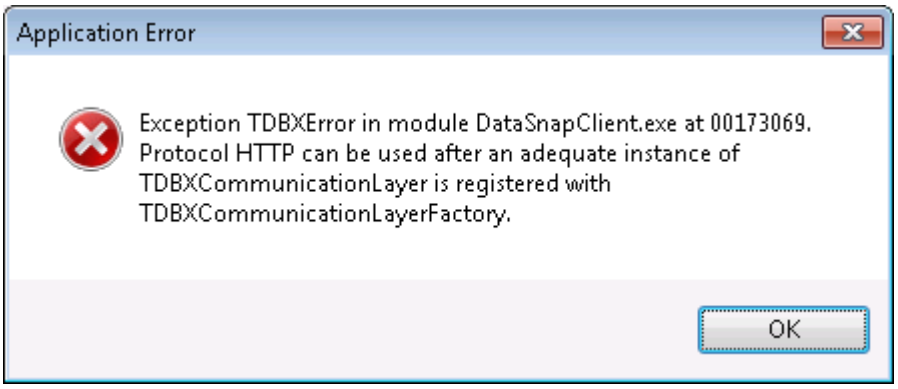


I leave it as exercise for the reader to test the EchoString method as well in a similar way.

2.2.1.1. HTTP COMMUNICATION PROTOCOL

Note that I mentioned the default CommunicationProtocol TCP/IP of the TSQLConnection component. This also means, we do not see any HTTP Trace messages (that we defined in section 2.1.1.1.4). However, it's not hard to change the communication protocol: just enter HTTP as value for the CommunicationProtocol sub-property of the TSQLConnection's Driver property. Note that this also means that you need to modify the Port property, since 211 was used for TCP/IP. Make sure to specify the same value for Port that was specified for the TDSHTTPService component in the ServerContainer.

After you've made these changes and run the DataSnap Client again, there's a good chance you will see the following application error:

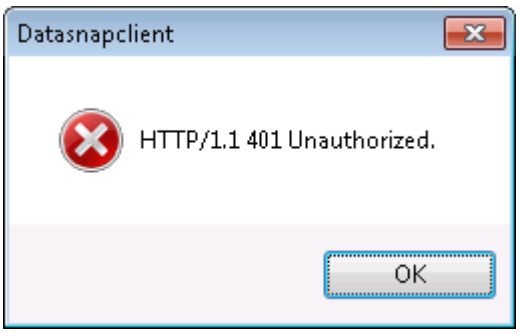


The solution for this error is to add the DSHTTPLayer unit to the uses clause (of the ClientForm for example) of the DataSnap Client.

2.2.1.2. HTTP AUTHENTICATION

One of the benefits of using the HTTP communication protocol is the ability to include HTTP Authentication. This is supported at the DataSnap Server side by the TDSHTTPServiceAuthenticationManager component, as covered in section 2.1.1.1.5.

As soon as the OnHTTPAuthenticate event handler is implemented, and a check for HTTP Authentication is made there, we need to ensure we're passing the right information in order to have the TDSHTTPServiceAuthenticationManager allow us access. Otherwise, you will get an HTTP/1.1 401 Unauthorized error:



To pass the HTTP Authentication username and password information from the DataSnap Client to the DataSnap Server, and more specifically the TDSHTTPServiceAuthentication component, we need to fill the DSAuthUser and DSAuthPassword properties of the TSQLConnection component (on the client form).

Note that we should also specify a value for the HostName here, unless we're testing with a DataSnap Server on the same local machine.

2.3. DataSnap Server Deployment

The example works fine when both the server and client are running on the same local machine, but in real-world situations, the DataSnap Server should be running on a server machine, with one or more clients connected to this server machine over the net. The machine where the DataSnap Server application is deployed to is most often a machine without Delphi installed. This means that it may be a consideration to compile the DataSnap Server without run-time packages enabled, so you get one big executable.
Since we haven't used any data-access components right now, there is no need for any additional database drivers or external DLLs at this time.

2.3.1. DataSnap Client Deployment

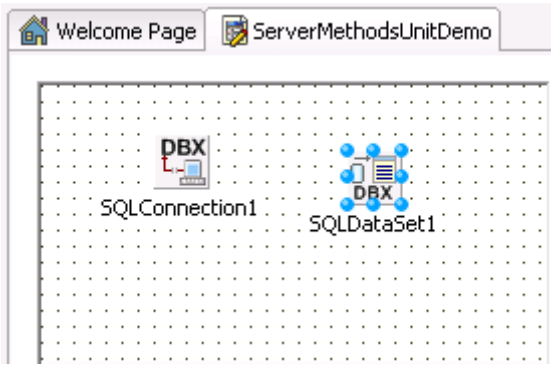
Assuming the DataSnap Client application is running on a different machine than the DataSnap Server application, we should make sure the client can connect to the server. In order to realize this, the TSQLConnection component on the client form should not only specify the values for the CommunicationProtocol and Port sub-properties of the Driver property, but also a value for the HostName. Try to avoid assigning an IP-address here, but use logical DNS name wherever possible. For my DataSnap servers, for example, I can use the www.bobswart.nl HostName value (note that you do not have to specify the http:// prefix, since the actual protocol used should be specified in the CommunicationProtocol).

3. DataSnap and Databases – where you want it

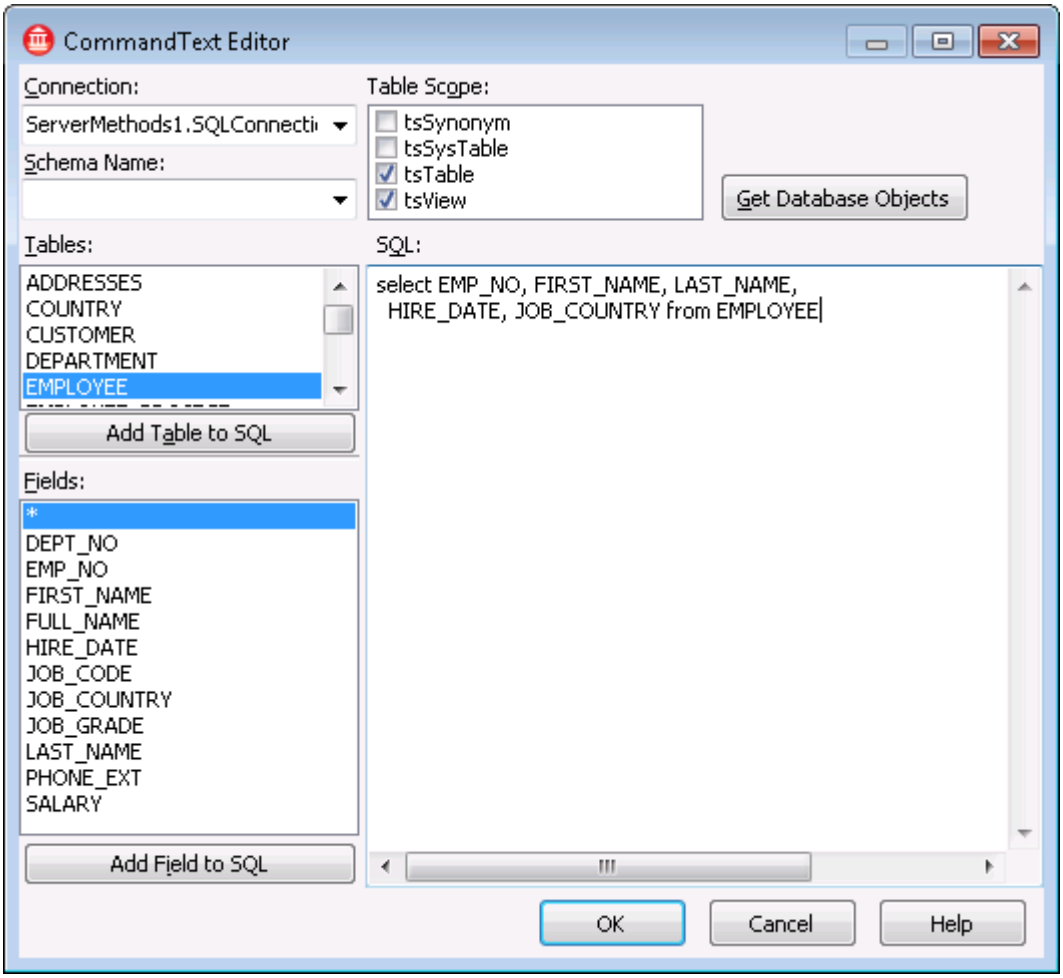
Apart from using the Delphi 2010 DataSnap framework to build simple server methods, we can also add database access to the server, turning the architecture in a multi-tier database application – where the DataSnap Server connects to the database, but the DataSnap Clients are thin or smart clients, without the need for a database driver (at the client side).

For the DataSnap example we've built so far, we've used only the SQLConnection component and the generated client classes directly, but we can also use the two new DataSnap components TsqlServerMethod and TDSProviderConnection.

First, we need to ensure that the server actually exposes a TDataSet, so return to the ServerMethodsUnitDemo and place a TSQLConnection component on the data module. Connect the TSQLConnection component to your favorite DBMS and table (in my example, I will connect to the Employees table from the BlackfishSQL Employees example database). In order to do that, place a TSQLConnection component on the ServerMethods1 designer area (in the ServerMethodsUnitDemo.pas unit). Set the Driver property of the TSQLConnection component to BlackfishSQL. Then, open up the Driver property and set the Database property to the path to the employee.jds in either C:\Documents and Settings\All Users\Documents\RAD Studio\7.0\Demos\database\databases\BlackfishSQL on Windows XP, or C:\Users\Public\Documents\ RAD Studio\7.0\Demos\database\databases\BlackfishSQL on Windows Vista or Windows 7. Make sure the LoginPrompt property of the TSQLConnection component is set to False, and then try to set the Connected property to True to see if you can open up the connection to the Employee.jds database.
Next, place a TSQLDataSet component next to the TSQLConnection component, and connect its SQLConnection property to the TSQLConnection component.



Leave the CommandType set to ctQuery, and double click on the ellipsis property for the CommandText property to enter an SQL query.




```
SELECT EMP_NO, FIRST_NAME, LAST_NAME, HIRE_DATE, JOB_COUNTRY FROM EMPLOYEE
```

Make sure the TSQLConnection component has its LoginPrompt and Connected property set to False at design-time, and ensure that the Active property of the TSQLDataSet is also set to False.

We should now add a public function to the TServerMethods1 class in the ServerMethodsUnitDemo unit to return the contents of the TSQLDataSet component.

```
type
  TServerMethods1 = class(TDSServerModule)
    SQLConnection1: TSQLConnection;
    SQLDataSet1: TSQLDataSet;
  private
    { Private declarations }
  public
    { Public declarations }
    function EchoString(Value: string): string;
    function ServerTime: TDateTime;
    function GetEmployees: TDataSet;
  end;
```

As you can see from the TServerMethod1 define, the new function is called GetEmployees and the implementation can be as follows:

```
function TServerMethods1.GetEmployees: TDataSet;
begin
  SQLDataSet1.Open; // make sure data can be retrieved
  Result := SQLDataSet1
end;
```

Recompile the server and make sure it's running again. Note that if you've closed down the DataSnap server, but are unable to recompile it, then it's most likely that the DataSnap server project is still running.

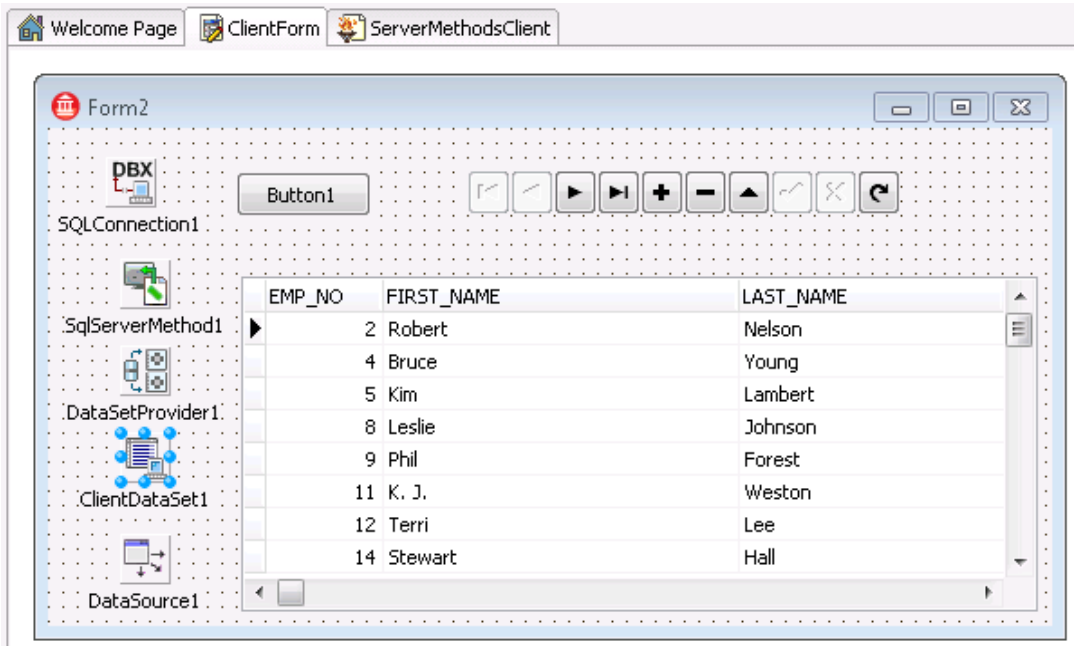
3.1. TSQLSERVERMETHOD

Return to the DataSnap client application. The TSQLConnection component should no longer be connected to the DataSnap Server (if it's still connected, then this may have been a reason why you couldn't recompile the DataSnap Server, since the process was still in use). We can set Connected to True again, to refresh the information from the server, and can then regenerate the client classes (using the right-mouse button again).
In order to overwrite the previous ServerMethodsClient unit, it's recommended to remove the previous version of ServerMethodsClient from the project and then select the "Generate DataSnap client classes" again. If we save the new generated unit in file ServerMethodsClient.pas again, then we don't have to change the previous client code.
After the "Generate DataSnap client classes" task has run again, the generated TServerMethods1Client class has been extended with the GetEmployees method (also listing the ServerTime and EchoString functions which are still available):

```
type
  TServerMethods1Client = class
  private
    FDBXConnection: TDBXConnection;
    FInstanceOwner: Boolean;
    FEchoStringCommand: TDBXCommand;
    FServerTimeCommand: TDBXCommand;
    FGetEmployeesCommand: TDBXCommand;
  public
    constructor Create(ADBXConnection: TDBXConnection); overload;
    constructor Create(ADBXConnection: TDBXConnection;
      AInstanceOwner: Boolean); overload;
    destructor Destroy; override;
    function EchoString(Value: string): string;
    function ServerTime: TDateTime;
    function GetEmployees: TDataSet;
  end;
```

In order to use the GetEmployees method to retrieve the TDataSet, we can use a TsqlServerMethod component from the dbExpress category of the Tool Palette. Place the TsqlServerMethod on the client form, connect its SQLConnection property to SQLConnection1, and then open up the ServerMethodName drop-down list to show all available methods we can call: a number of DSAdmin methods (which can be disabled by setting the HideDSAdmin property of the TDSServer component to True), followed by 3 DSMetadata methods, 7 TServerMethods.AS_xxx methods (offering the original IAppServer interface) and finally our TServerMethods1 EchoString, ServerTime and GetEmployees methods.
For the current example, we need to select the TServerMethods1.GetEmployees as value for the ServerMethodName property. With this value for the ServerMethodName, the result of the SqlServerMethod is a dataset with the employees records inside (or at least with the result from our SQL statement).
We can use the usual TDataSetProvider, TClientDataSet and TDataSource chain now to enable the data to be shown in a TDBGrid for example.
Place a TDataSetProvider on the client form and connect its DataSet property to the SqlServerMethod component. Next, place a TClientDataSet component on the client form and connect its ProviderName property to the DataSetProvider component. Leave the RemoteServer property blank – this was only used in the "old" DataSnap approach, but no longer in the new DataSnap architecture.

Finally, place a TDataSource on the client form and connect its DataSet property to the TClientDataSet component, and after that we can use a TDBGrid and TDBNavigator component, connect their DataSource property to the TDataSource component and be able to view the data in the client form.
In order to verify the connection at design-time, we can set the ClientDataSet's Active property to True. This will toggle the Active property of the SqlServerMethod (only for a short while to retrieve the data, after which the Active property it set to False again), which will set the Connected property of the TSQLConnection component to True to enable the connection between the DataSnap client and the server.
The connection remains active, by the way, and the result is both the TClientDataSet and the TSQLConnection being active, showing the data inside the TDBGrid:



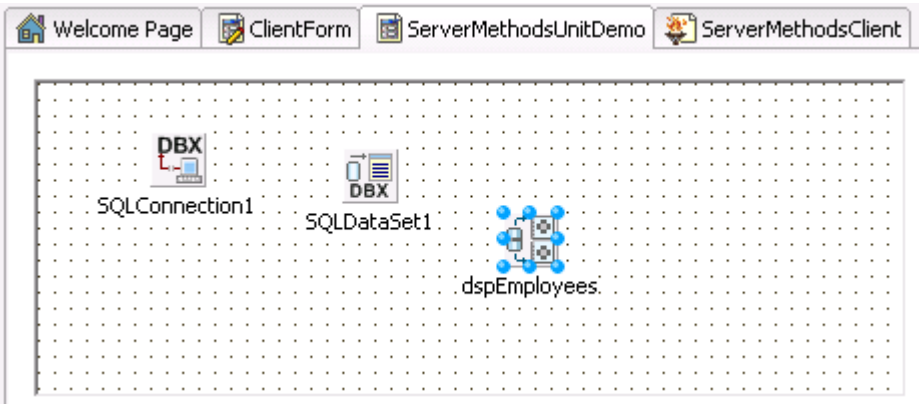
This provides an easy way to view the data and look at it in a read-only way. Note that I specifically say "read-only", since the TSQLServerMethod does not allow the TDataSetProvider-TClientDataSet combination to send any updates from the DataSnap client to the DataSnap server this way.

The TSQLServerMethod is a light-weight an convenient way to connect to read-only data. This means that if you want to expose data from a DataSnap Server that people should never be able to modify, then the current architecture, exposing TSQLDataSet results from the TServerMethods1 class is a good idea.
However, sometimes we need to allow updates, in which case we must use a different approach to expose the data.

3.2. TDSProviderConnection

If we want to apply updates, then we need for example a TDSProviderConnection component which gives us a reference to a DataSetProvider at the server side, so we can not only read the data, but also send the updates.
First, we need to make a change to the server data module at the server side, however, since right now we only added a function to return the TDataSet, but now we must add an actual TDataSetProvider and make sure that's exported from the DataSnap server to the DataSnap clients. So, return to the ServerMethodsUnitDemo unit and place a TDataSetProvider component next to the TSQLDataSet, pointing the DataSet property of the

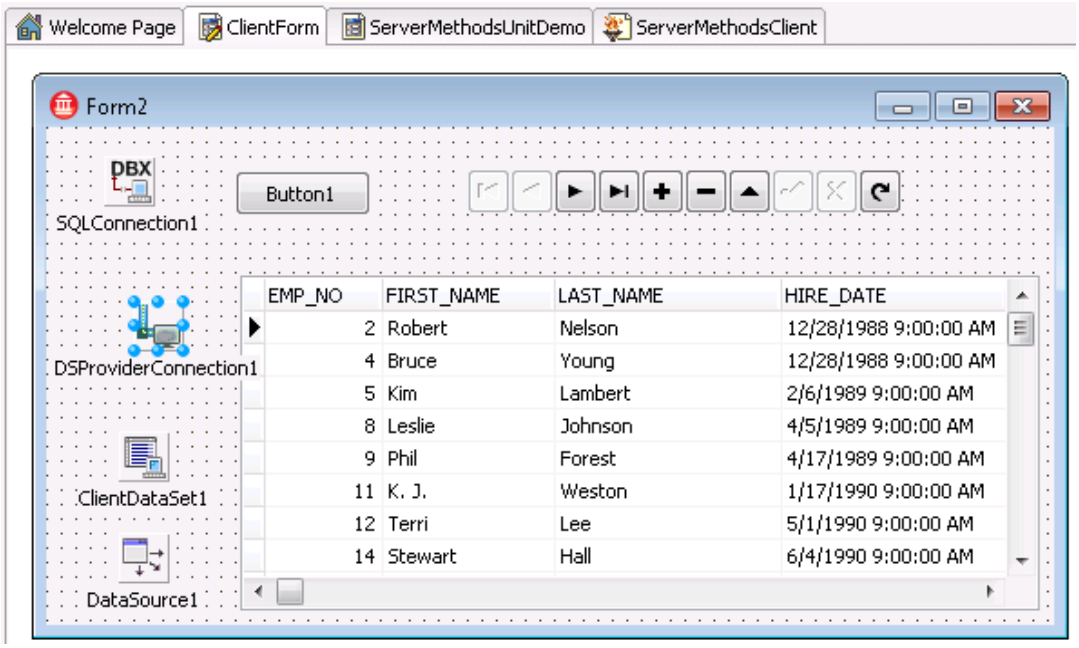
TDataSetProvider to the TSQLDataSet. We should also rename the TDataSetProvider component, making sure that it has a meaningful like dspEmployees.



Now, recompile the DataSnap server and run it again, so we can modify the client in order to make changes to the exposed data.

3.2.1. TDSProviderConnection Client

To change the DataSnap Client in order to make use of the exposed TDataSetProvider component, we have to remove the TSQLServerMethod and TDataSetProvider components from the client form, but we need to add a TDSProviderConnection component instead. Point the SQLConnection property of the TDSProviderConnection to the TSQLConnection component, connecting to the DataSnap Server. We also need to enter a value for the ServerClassName property of the TDSProviderConnection. It's a bit unfortunate that we (still) do not get a drop-down list here. Right now, we must manually enter the name of the TDSServerModule, which was TServerMethods1 in our case.
In the previous example, the TClientDataSet only connected its ProviderName to the DataSetProvider1. However, using the TDSProviderConnection component, we must first assign the RemoteServer property of the TClientDataSet to the TDSProviderConnection component, and then select a new value for the ProviderName property (which was still assigned to the DataSetProvider1 value by the way, but should now point to dsEmployees – the more descriptive name of the TDataSetProvider component which was exposed from the DataSnap Server).
The drop-down combobox for the ProviderName property should now show the dspEmployees option (the name of the TDataSetProvider component which is exported from the ServerDataMod unit).
Now, we can set the Active property of the TClientDataSet component to True again in order to view live data at design-time:



Setting the Active property of the TClientDataSet component will also set the Connected property of the TSQLConnection component to True. Note that it's not a good idea to leave these two properties set to True in the client form at design-time. First of all, if you open the DataSnap Client project in the Delphi IDE, it will attempt to make a connection to the DataSnap Server, which may fail if the server is not up-and-running. Second, if you start the application at run-time, and no connection is available, then the application will also fail with an exception. This prevents you from using the application on local data, rendering it useless if no connection is present.

A better way is to include some menu option or button to explicitly connect the TSQLConnection component and/or activate the TClientDataSet. This is also a great moment to include username/password information, which we'll cover later. For now, make sure the Active property of the TClientDataSet is set to False, as well as the Connected property of the TSQLConnection component. Then, place a button on the client form, and in the OnClick event handler open the TClientDataSet explicitly.

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  ClientDataSet1.Open;
end;
```

Time to add some code to actually change the data (at the client side), and send the changes back to the server.

3.2.2 Database Updates

There are actually two ways to send the changes we've made at the client back to the server: automatic or manual. Both call the same method in the end, but the invocation is either automatic or user-driven, both with advantages and disadvantages.

For the automatic approach, we can use the OnAfterInsert, OnAfterPost and OnAfterDelete events of the TClientDataSet, since these are the methods that have made changes to the data. In the event handlers – which can be shared in a single implementation – we can call the ApplyUpdates method of the TClientDataSet, sending the changes, also called "Delta" back to the server to be resolved back in the database.

```
procedure TForm2.ClientDataSet1AfterPost(DataSet: TDataSet);
```

```
begin
  ClientDataSet1.ApplyUpdates(0);
end;
```

If something bad has happened (like a record no longer found) during the update, we can get feedback in the OnReconcileError event of the TClientDataSet, which is covered in more detail in section 3.2.3.

The manual way of sending the updates back to the DataSnap Server also makes use of the TClientDataSet's ApplyUpdates method, but this time the method should not be called in the OnAfterInsert, OnAfterPost and OnAfterDelete event handlers. Instead, we should add a button on the client form to allow the user to explicitly post the updates back to the server.

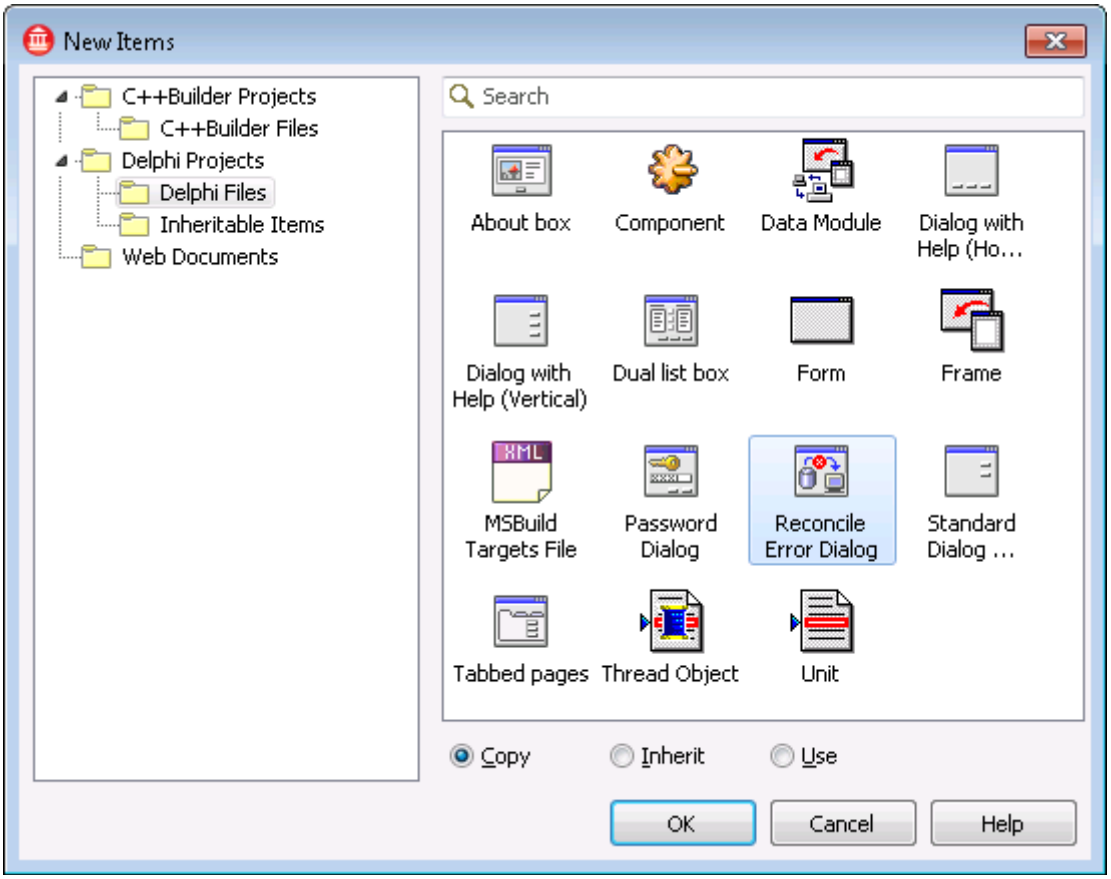
```
procedure TForm2.btnUpdateClick(Sender: TObject);
  ClientDataSet1.ApplyUpdates(0);
end;
```

The advantage of doing the automatic call to ApplyUpdates is of course that the user will never "forget" to apply any changes back to the server. However, the disadvantage is that there is no undo possibility: once posted, the data is applied to the server. On the other hand, if the manual approach is used, then all changes are kept at the client side – inside the memory of the TClientDataSet component. This allows the user to undo certain parts of the changes: either the last change, a specific record or the entire pending updates. Clicking on the "update" button to explicitly call the ApplyUpdates methods when the user is ready. The possible danger is that the user could "forget" to click on the update button, so we should add a check to the form or application to prevent it from closing when there are still changes left in the TClientDataSet. The latter can be checked by looking at the ChangeCount property of the TClientDataSet.

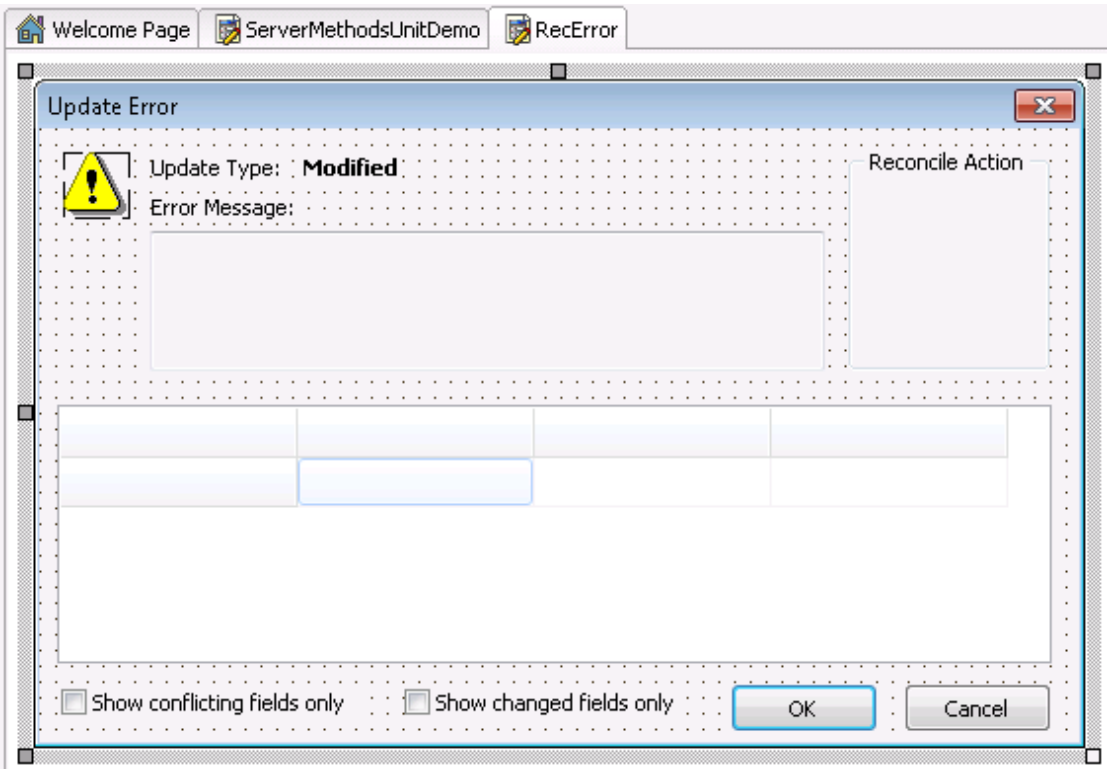
3.2.3. RECONCILE ERRORS

The ApplyUpdates method of the TClientDataSet component has one argument: the maximum number of errors that it will "allow" before stopping with applying (more) updates. So what if two clients connect to the DataSnap Server, obtain the Employees data and both make some changes to the first record. According to what you've build so far, both clients could then send the updated record back to the DataSnap Server using the ApplyUpdates method of their TClientDataSet component. If both pass zero as value for the "MaxErrors" argument of ApplyUpdates, then the second one to attempt the update will be stopped. The second client could pass a numerical value bigger than zero to indicate a fixed number of errors/conflicts that are allowed to occur before the update is stopped. However, even if the second client passed -1 as argument (to indicate that it should continue updating no matter how many errors occur), it will never update the records that have been changed by the previous client. In other words: you need to perform some reconcile actions to handle updates on already-updated records and fields.

Fortunately, Delphi contains a very useful dialog especially written for this purpose. And whenever you need to do some error reconciliation, you should consider adding this dialog to your DataSnap Client application (or write one yourself, but at least do something about it).
To use the one available in Delphi, just do *File* | *New - Other*, go to the Delphi Files subcategory of the Delphi Projects in the Object Repository and select the Reconcile Error Dialog icon.



Once you select this icon and click on OK, a new unit RecError.pas is added to your DataSnapClient project. This unit contains the definition and implementation of the Update Error dialog that can be used to resolve database update errors.



An instance of the ReconcileErrorForm will be created dynamically, on-the-fly, when it is needed. *So when or how do you use this special ReconcileErrorForm?* Well, it's actually very simple. For every record for which the update did not succeed (for whatever reason), the OnReconcileError event handler of the TClientDataSet component is called. This event handler of TClientDataSet is defined as follows:

```
procedure TForm2.ClientDataSet1ReconcileError(DataSet: TClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind;
  var Action: TReconcileAction);
```

This is an event handler with four arguments: first of all the TClientDataSet component that raised the error, second a specific ReconcileError that contains a message about the cause of the error condition, third the UpdateKind (insert, delete or modify) that generated the error and finally as fourth argument the Action that you feel should be taken.
As Action, you can return the following possible enum values (the order is based upon their actual enum values):

- raSkip - do not update this record, but leave the unapplied changes in the change log. Ready to try again next time.
- raAbort - abort the entire reconcile handling; no more records will be passed to the OnReconcileError event handler.
- raMerge - merge the updated record with the current record in the (remote) database, only changing (remote) field values if they changed on your side.
- raCorrect - replace the updated record with a corrected value of the record that you made in the event handler (or inside the ReconcileErrorDialog. This is the option in which user intervention (i.e. typing) is required.
- raCancel - undo all changes inside this record, turning it back into the original (local) record you had.
- raRefresh - undo all changes inside this record, but reloading the record values from the current (remote) database (and not from the original local record you had).

The good thing about the ReconcileErrorForm is that you don't really need to concern yourself with all this. You only need to do two things. First, you need to include the ErrorDialog unit inside the DataSnap Client main form definition. Click on the DataSnap Client Form and do *File | Use Unit* to get the Use Unit dialog. With the Client Form as your current unit, the Use Unit dialog will list the only other available unit, which is the ErrorDialog. Just select it and click on OK.
The second thing you need to do is to write one line of code in the OnReconcileError event handler in order to call the HandleReconcileError function from the ErrorDialog unit (that you just added to your ClientMainForm import list). The HandleReconcileError function has the same four arguments as the OnReconcileError event handler (not a real coincidence, of course), so it's a matter of passing arguments from one to another, nothing more and nothing less. So, the OnReconcileError event handler of the TClientDataSet component can be coded as follows:

```
procedure TFrmClient.ClientDataSet1ReconcileError(DataSet: TClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind;
  var Action: TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E)
end;
```

3.2.4. DEMONSTRATING RECONCILE ERRORS

The big question now is: *how does it all work in practice?* In order to test it, you obviously need two (or more) DataSnap Client applications running simultaneously. For a complete test using the current DataSnap Client and DataSnap Server applications, you need to perform the following steps:

- Start the DataSnap Server application.
- Start the first DataSnap Client, and click on the Connect button.
- Start the second DataSnap Client and click on the Connect button. Data will be obtained from the same DataSnap Server that's already running.
- Using the first DataSnapClient, change the field "FirstName" for the first record.
- Using the second DataSnap Client, also change the field "FirstName" for the first record.
- Click on the "Apply Updates" button of the first DataSnap Client. All updates will be applied without any problems.
- Click on the "Apply Updates" button of the second DataSnap Client. This time, one or more errors will occur, because the first record has its "FirstName" field value changed (by the first DataSnap Client), and for this and possibly more conflicting records, the OnReconcileError event handler is called.
- Inside the Update Error dialog, you can now experiment with the reconcile Actions (skip, abort, merge, correct, cancel and refresh) to get a good feeling of what they do. Pay special attention to the differences between Skip and Cancel, and the differences between Correct, Refresh and Merge.

Skip moves on to the next record, skipping the requested update (for the time being). The unapplied change will remain in the change log. *Cancel* also skips the requested update, but it cancels all further updates (in the same update packet). The current update request is skipped in both cases, but Skip continues with other update requests, and Cancel cancels the entire ApplyUpdates request.
Refresh just forgets all updates you made to the record and refreshes the record with the current value from the server database. *Merge* tries to merge the update record with the record on the server, placing your changes inside the server record. Refresh and Merge will not process the change request any further, so the records are synchronized after Refresh and Merge (while the change request can still be redone after a Skip or Cancel).
Correct, the most powerful option, actually gives you the option of customizing the update record inside the event handler. For this you need to write some code or enter the values in the dialog yourself.

3.3. DATASNAP "DATABASE" DEPLOYMENT

Deployment of a DataSnap Server that uses databases can be a bit more involved than the deployment of the simple DataSnap Server we started with. For the client application, nothing changes – it's still a thin / smart client application, and can be deployed as stand-alone executable if you add the MidasLib unit to the uses clause.

For the DataSnap Server, we must now also deploy the database drivers. Which drivers and files depends on the database you selected. When using DBX4, make sure to check the TSQLConnection component as well as the dbxconnections.ini and dbxdrivers.ini files which can be found in the C:\Documents and Settings\All Users\ Documents\RAD Studio\dbExpress\7.0 directory on Windows XP or in the C:\Users\Public\Documents\RAD Studio\dbExpress\7.0 directory on Windows Vista and Windows 7.
The dbxdrivers.ini file will specify – for the given Driver – the DriverPackageLoader and MetaDataPackageLoader (usually pointing to the same package). For BlackfishSQL, this means DBXClientDriver140.bpl, which should be deployed as well as Blackfish itself. For more information on BlackfishSQL deployment, see the file deploy_en.htm in the RAD Studio\7.0 directory.

3.4. Reusing Existing Remote Data Modules

If you have existing TRemoteDataModule classes, then you can still use these in combination with the new DataSnap. But you have to cut some functionality from the server, especially the COM-stuff.
First of all, if it's an existing DataSnap Server application that you want to migrate, and not just the remote data module, you need to unregister the DataSnap server by running the executable from the command-line with the /unregister command-line option. If you don't do that right from the start, you will not be able to unregister the remote data module from the registry (unless you can restore a backup of the project later).
In the unit for the remote data module, we must remove the code from the initialization section. If you want to keep your unit compatible between Delphi 2007-or-below, and 2009-or-later, you can place this code inside {$IFDEF}s as follows:

```
{$IF CompilerVersion >= 20}
initialization
  TComponentFactory.Create(ComServer, TRemoteDataModule2010,
    Class_RemoteDataModule2010, ciMultiInstance, tmApartment);
{$IFEND}
end.
```

We should also remove the UpdateRegistry routine from the project, or place it in {$IFDEF}s as well.

```
{$IF CompilerVersion >= 20}
  class procedure UpdateRegistry(Register: Boolean;
    const ClassID, ProgID: string); override;
{$IFEND}
```

The most important change – to turn the project into a COM-less DataSnap server – involves the removal of the type library (or .ridl files) and the type library import unit. These cannot be left in {IFDEF}s, so if you need to keep a Delphi 2007 or below (COM-enabled) and Delphi 2009 or later (COM-less) version of the DataSnap server you need to make a copy of the project now. We should use a TDSServerClass component in the DataSnap server application and return the TRemoteDataModule class, just as we've done before.
Finally, we should make sure that all custom methods that were added to the TRemoteDataModule are moved from the protected section (the default in COM-enabled DataSnap) to the public section (so method info is generated in the COM-less DataSnap architecture).

4. DataSnap Filters – how you want it

In this section I will explain how filters work, and how we can use existing filters (such as compression) or build new DataSnap filters ourselves. DataSnap Filters are special DLLs that intercept the communication byte stream, and can actually operate in a whole chain of filters. So we can combined compression and encryption for example, or logging with compression, etc.

There are two places where we must specify which filter(s) the DataSnap Server and Client should use. For the Server, we must specify a list in the Filters property of the TDSTCPServerTransport component. And for the Clients, we must specify a similar list of filters in the uses clause of the DataSnap Client project. This is enough for the client, since each DataSnap filter should automatically register itself for use.

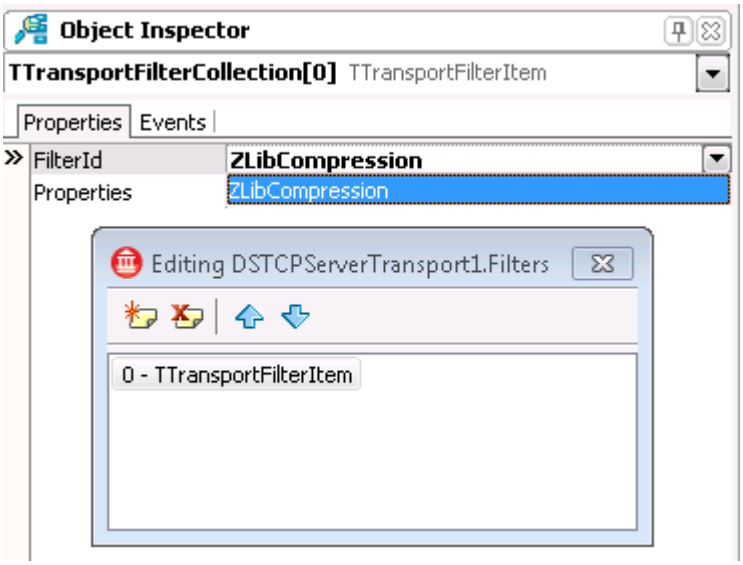
During the OnConnect event handler, we can examine the registered filters that are used for the connection, for example (using a custom function LogInfo to write this information to a logfile):

```
procedure TServerContainer1.DSServer1Connect(
  DSConnectEventObject: TDSConnectEventObject);
var
  i: Integer;
begin
  LogInfo('Connect ' + DSConnectEventObject.ChannelInfo.Info);
  for i:=0 to DSConnectEventObject.Transport.Filters.Count-1 do
    LogInfo('  Filter: ' +
      DSConnectEventObject.Transport.Filters.GetFilter(i).Id);
end;
```

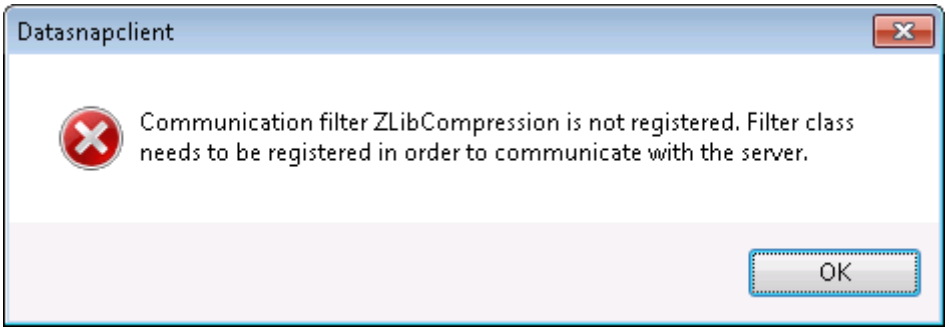
4.1. ZLIBCOMPRESSION FILTER

As an example, let's examine an existing DataSnap Filter, which ships with Delphi 2010 already, and can be used to compress the data stream between the DataSnap Server and Client (and vice versa). I'm talking about the ZlibCompression filter, which can be found in the DbxCompressionFilter unit.

Both the TDSTCPServerTransport component (for TCP/IP) and the DSHTTPService component (for HTTP) have a Filters property that holds a TTransportFiltersCollection. We can click on the ellipsis for the Filters property to edit the collection of filters. In this dialog, we can add a new TTransportFilterItem, and then use the Object Inspector to set the FilterId and some optional properties. Out-of-the-box, Delphi 2010 comes with the ZLibCompression filter, which can be specified as FilterId here.



Note that apart from the Filters property at the TDSTCPServerTransport component at the server side, we should also specify that we want to use this filter at the client side (to compress the outgoing requests and decompress the incoming responses). For this, we only need to add the DbxCompressionFilter unit to the uses clause of the ClientForm. That will automatically register the TTransportCompressionFilter and make sure it's used to communicate with the server.
If you do not add the DbxCompressionFilter unit to the uses clause, then running the client will raise an exception with the message *"Communication filter ZLibCompression is not registered. Filter class needs to be registered in order to communicate with the server."*.



4.2. LOG FILTER

Delphi 2010 DataSnap is open to allow us to define our own transport filters. We can do this by deriving a new class from the TTransportFilter type. In this new class, we can override the base methods, and implement them. As an example, we can create a TLogFilter class as follows:

```
unit LogFilter;
interface
uses
  SysUtils, DBXPlatform, DBXTransport;

type
  TLogFilter = class(TTransportFilter)
  private
  protected
    function GetParameters: TDBXStringArray; override;
    function GetUserParameters: TDBXStringArray; override;
  public
    function GetParameterValue(const ParamName: UnicodeString): UnicodeString; override;
    function SetParameterValue(const ParamName: UnicodeString;
      const ParamValue: UnicodeString): Boolean; override;
    constructor Create; override;
    destructor Destroy; override;
    function ProcessInput(const Data: TBytes): TBytes; override;
    function ProcessOutput(const Data: TBytes): TBytes; override;
    function Id: UnicodeString; override;
  end;

const
  LogFilterName = 'Log';
```

The implementation of this class can be left empty on most places: since the only purpose of the log filter is to log the data which is sent during the ProcessInput and ProcessOutput methods, we can leave most of the other methods empty. The implementation of the non-empty methods is as follows:

```
function TLogFilter.SetParameterValue(const ParamName, ParamValue: UnicodeString): Boolean;
begin
  Result := True;
end;

constructor TLogFilter.Create;
begin
  inherited Create;
end;

destructor TLogFilter.Destroy;
begin
  inherited Destroy;
end;

function TLogFilter.ProcessInput(const Data: TBytes): TBytes;
begin
  Result := Data; // log incoming data
```

```
end;

function TLogFilter.ProcessOutput(const Data: TBytes): TBytes;
begin
  Result := Data; // log outgoing data
end;

function TLogFilter.Id: UnicodeString;
begin
  Result := LogFilterName;
end;
```

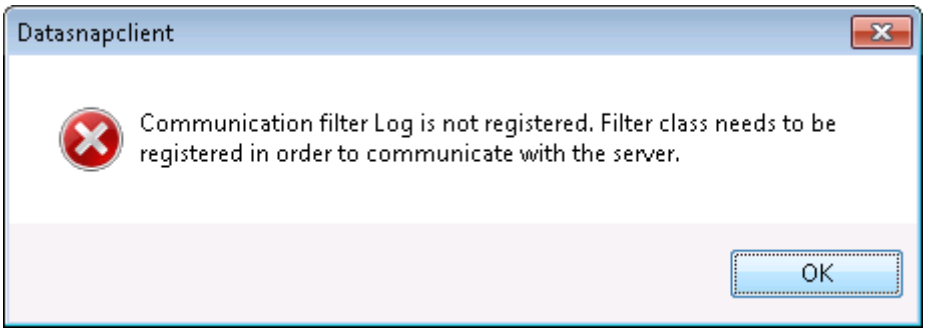
Finally, an important part of the implementation of a DataSnap transport filter is the registration part in the initialization and finalization section. This makes sure that the DataSnap client can "find" the transport filter and use it when required.

```
initialization
  TTransportFilterFactory.RegisterFilter(LogFilterName, TLogFilter);
finalization
  TTransportFilterFactory.UnregisterFilter(LogFilterName);
end.
```

In order to use the transport filter in the DataSnap Server, we have to add it to the list of Filter of the TDSTCPServerTransport or the TDSHTTPService component, as discussed earlier. At design-time, the ZLibCompression filter is already known, but not any new filter (unless we add them in a design-time package and install them). Fortunately, we can also add the transport filters at run-time, by adding the filter unit to the uses clause of the ServerContainerUnitDemo, and then manually adding the filter (by name) to the list of filters, for example in the

```
procedure TServerContainer1.DataModuleCreate(Sender: TObject);
begin
  DSTCPServerTransport1.Filters.AddFilter(LogFilterName);
  DSHTTPService1.Filters.AddFilter(LogFilterName);
  DSHTTPService1.Active := True;
end;
```

This will ensure that the server is using the LogFilter, and the client will be using it automatically after the LogFilter unit is added to the uses clause of the client. Otherwise, the following error message will be shown:



Note that each application – DataSnap Server and Clients – will get its own logfile, so although the same logging filter is used, we do not have to add information like ParamStr(0) about which target is actually producing the log message.

4.3. Encryption Filter

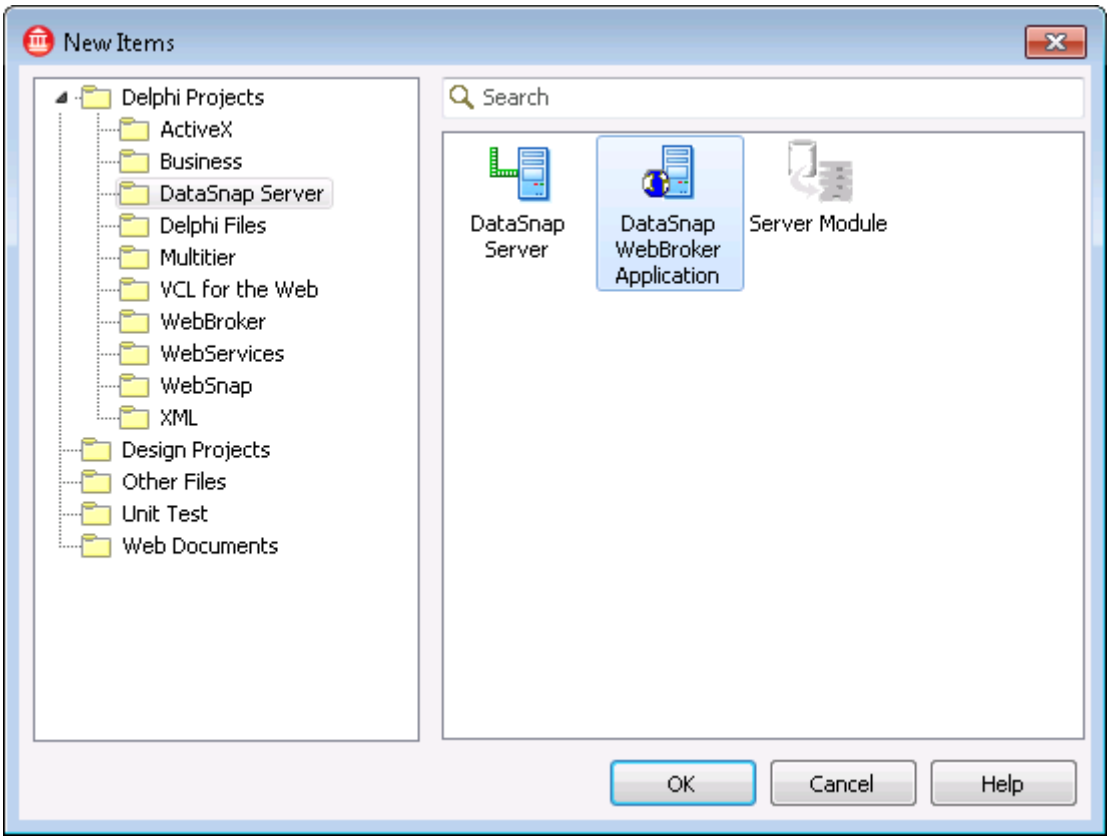
Given the simple filter example from section 4.2, it should be clear that expanding on this example and building your own, more complex, DataSnap Filters is not really that complicated. In fact, a number of third-party filters are already available, and a DataSnap

Filters Compendium by Daniele Teti can be found at http://www.danieleteti.it/?p=168 containing no less than 9 additional filters for DataSnap 2010, divided into three groups. The Hash group supports MD5, MD4, SHA1 and SHA512, the Cipher group supports Blowfish, Rijndael, 3TDES and 3DES, and the Compress group supports LZO. Full source code is available.

5. DataSnap Web Targets – how you want it (more)

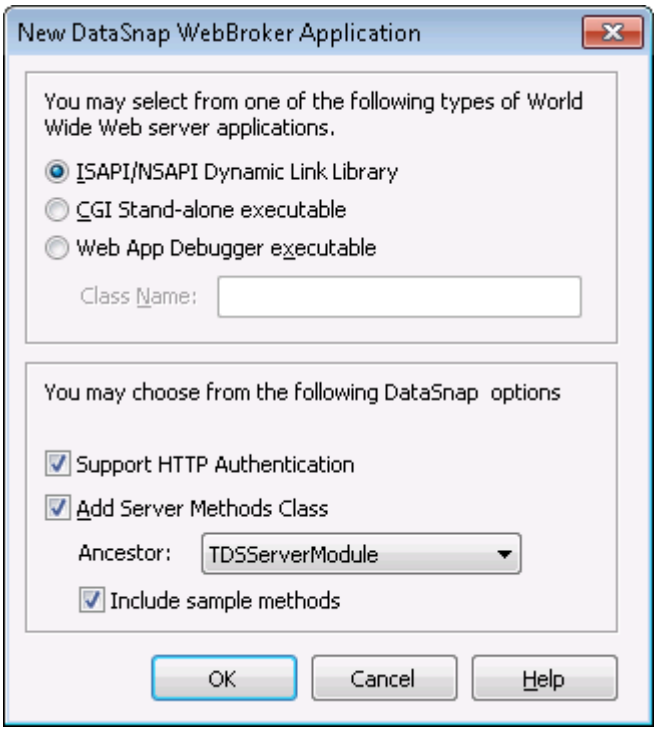
Apart from Windows targets, there is also a wizard to produce ISAPI, CGI or Web App Debugger targets. I'll first discuss the benefits of each of these targets, and also show how we can produce a single project group with three different targets that sill share the DataSnap custom units between them, so the result is a single project group with three projects that just produce a different target for the same DataSnap server object.

Although the DataSnap server applications that we've built so far work fine, there are moments where you are not able to deploy these server applications. For example because you are unable or not allowed to open up the required ports on the firewall to allow clients to connect to the server. Fortunately, in most situations where this is the case, there will be a web site hosted on a web server, so port 80 is generally open (for the web server, that is). And if we assume for a moment that Microsoft Internet Information Services (IIS) is used as web server, then we can use the new DataSnap WebBroker Application wizard to produce a project that can be deployed on IIS.



The DataSnap WebBroker Application wizard will offer three choices, one of which is actually not a real WebBroker application, but merely a web app debugger client that should only be used for debugging purposes. The Web App Debugger client is very powerful, since it allows

us to use the Web App Debugger (available from the Tools menu of the Delphi IDE) as host application while debugging the Web App Debugger Client DataSnap application. Debugging a CGI or ISAPI/NSAPI web application is far less easy. So Web App Debugger is a powerful choice while the application is still in development.
The remaining ISAPI/NSAPI Dynamic Link Library and CGI Stand-alone executable targets can be selected for real DataSnap server projects.

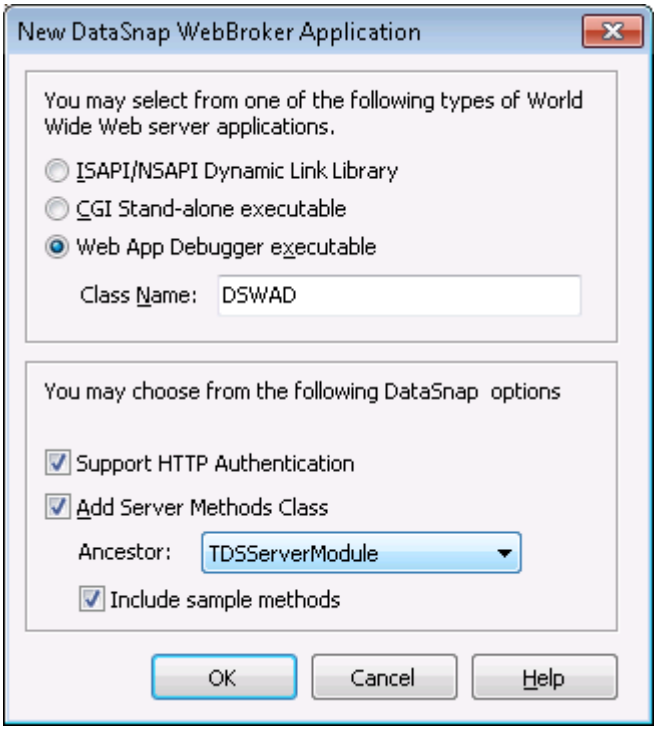


Note, however, that selecting a CGI Stand-alone executable is not a very good idea because this executable will be loaded and unloaded for each incoming requests. Add the time to connect to a database to perform some work, and you may get an idea about the performance hit this application type will suffer. Using an ISAPI target will result in a DLL which is loaded only once, and remains loaded in memory so subsequent requests (also from other users) do not suffer from an additional performance penalty. The major downside of an ISAPI DLL is that it's not easy to replace one (if you only have FTP access to the web server), but there are enough ISAPI Managers out there to solve this task for you (contact your web host provider for details).

Another downside of the ISAPI DLL target is that it's not easy to debug – you have to load IIS as host application, which doesn't always work as planned. But that particular issue is solved by the presence of the Web App Debugger executable – you only have to make sure to use two projects, both using the same actual DataSnap custom methods and code. Which is a good start for a first demo, adding some actual real-world techniques to ensure a working skeleton.

5.1. Web App Debugger Target

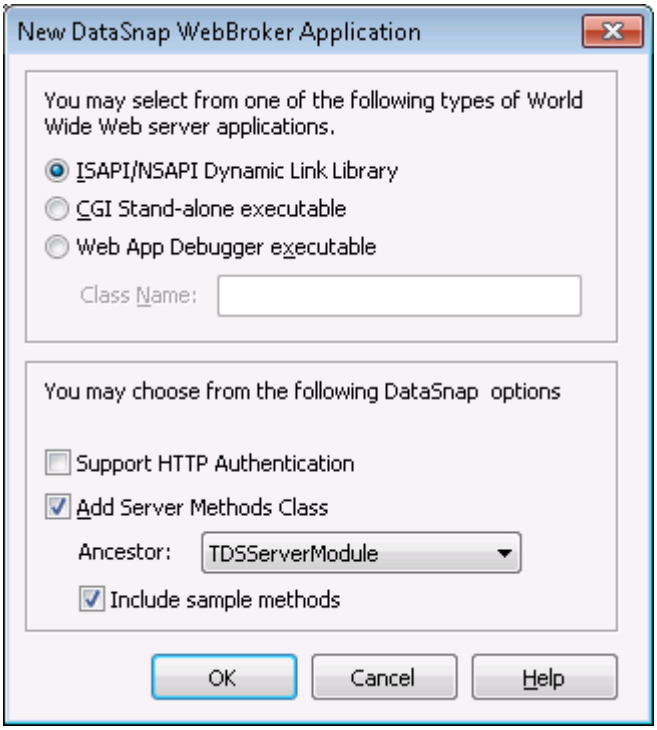
First, use the New DataSnap WebBroker Application wizard to create a new Web App Debugger application. Specify something for the Class Name, such as DS-WAD and check the "Support HTTP Authentication" option.



Once you click on OK, a new project will be created, with three units. If you don't have any Project1 and Unit1 files in your default projects directory, the project will be named Project1, and the units will be called Unit1, ServerMethodsUnit1 and Unit2 respectively. The first unit should be an empty form – this one is unique for the Web App Debugger executable, and not needed for the other web targets of course. Save this unit in WADForm.pas. The second unit is called ServerMethodsUnit1.pas and contains our server module, derived from TDSServerModule as specified in the dialog. We'll get back to this unit in a moment, but we can save it using the given name ServerMethodsUnit1.pas for now. The third unit is named Unit2, and should be a web module with four components already present on it (three if you didn't check the "Support HTTP Authentication" option). This is the unit to receive the incoming requests and distribute them over the DataSnap server modules in the project. Save this unit in DSWebMod.pas.
Finally, save the project as DSWADServer.dproj to indicate that this project is the DataSnap Web App Debugger Server.

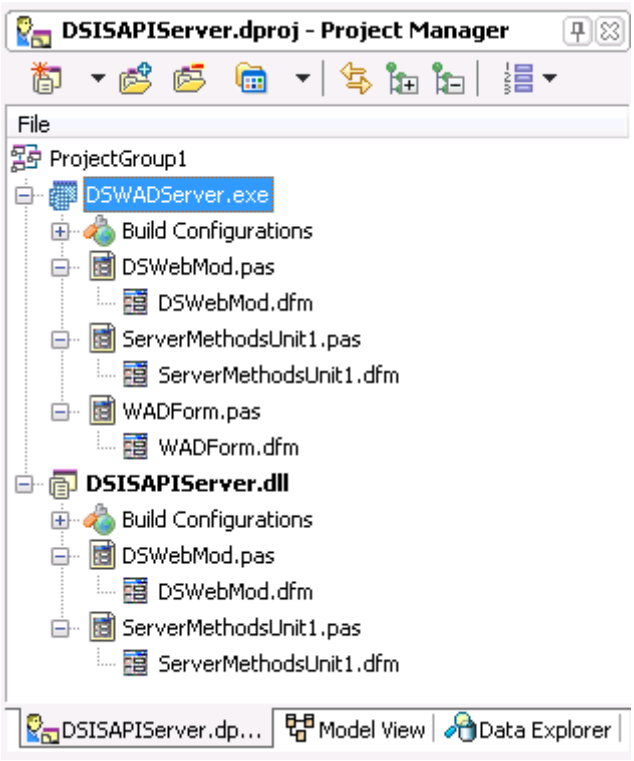
5.2. ISAPI TARGET

Before we continue and modify and customize the ServerMethodsUnit1.pas and DSWebMod.pas units, we should first add a new project to the project group, this time an ISAPI/NSAPI Dynamic Link application. So, right-click on the ProjectGroup and select Add New Project to get the Object Repository in order to start a new project. From the DataSnap Server category, use the New DataSnap WebBroker Application wizard again to create a new ISAPI/NSAPI Dynamic Link application. This time, you do not have to modify any option from the bottom part of the dialog, since we're going to re-use the existing units from the DSWADServer project.



Click on OK to generate a new project (which will be added to the project group), again called Project1, with units ServerMethodUnit2.pas and Unit1.pas added to the new project.

Now, instead of using the new units ServerMethodUnit2.pas and Unit1.pas, we should use the units ServerMethodUnit1.pas and DSWebMod.pas that are already part of the DSWADServer project. So, right-click on the ServerMethodUnit2.pas node under the Project1.dll node and select Remove from Project. Click OK in the confirmation dialog (note that if you didn't save them yet, you do not have to delete the ServerMethodUnit2.pas and .dfm files from disk). Do the same with Unit1.pas, so the Project1.dll contains no more units now. Then, right-click on Project1.dll and add both units DSWebMod.pas and ServerMethodUnit2.pas to this project. Finally, rename Project1 to DSISAPIServer.dproj to complete the project group.
You should now have one project group with two projects, sharing the DSWebMod and DSSererMethodUnit1.pas units, as can be seen in the following screenshot:



This setting will allow you to build two projects, using DSWADServer as target for your testing and debugging, and DSISAPIServer for the actual deployment of the DataSnap server on IIS.

Before we continue by adding web methods to the ServerMethodsUnit1, we should first "fix" the ISAPI/NSAPI project by removing from project file the code that creates an automatic instance of the TDSServerModule. Since a TDSServerModule is ultimately also a data module, just like the web module, we'll get an error message when trying to run the ISAPI DLL, since there can be only one web module in the web broker application.
Open the DSISAPIServer.dpr project source code, and change the main begin-end block as follows:

```
begin
  CoInitFlags := COINIT_MULTITHREADED;
  Application.Initialize;
  Application.CreateForm(TWebModule2, WebModule2);
//  Application.CreateForm(TServerMethods1, ServerMethods1);
  Application.Run;
end.
```

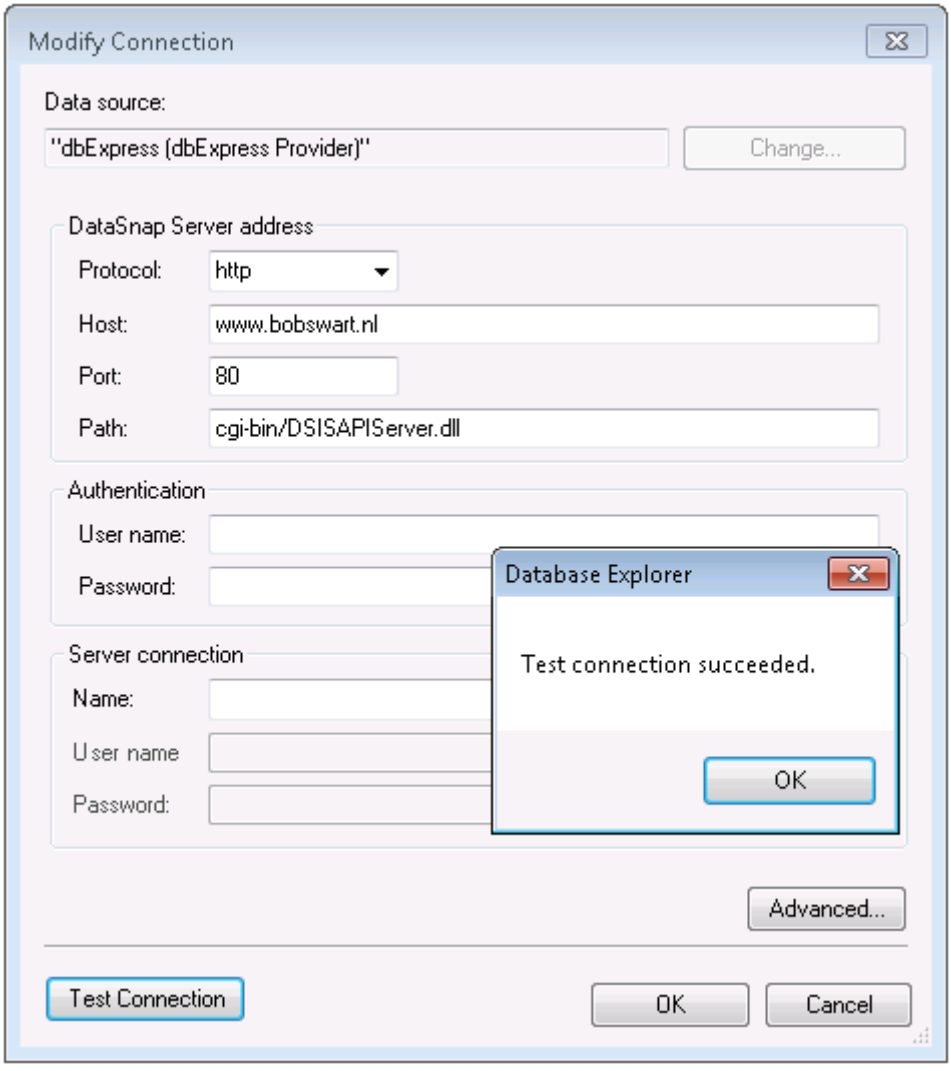
This will avoid the error message that only one data module is allowed. Note that you may not see this error message when you actually call the (deployed) ISAPI DLL, but merely a server error or time-out, so it's important to remember this issue.

5.3. Server Methods, Deployment and Clients

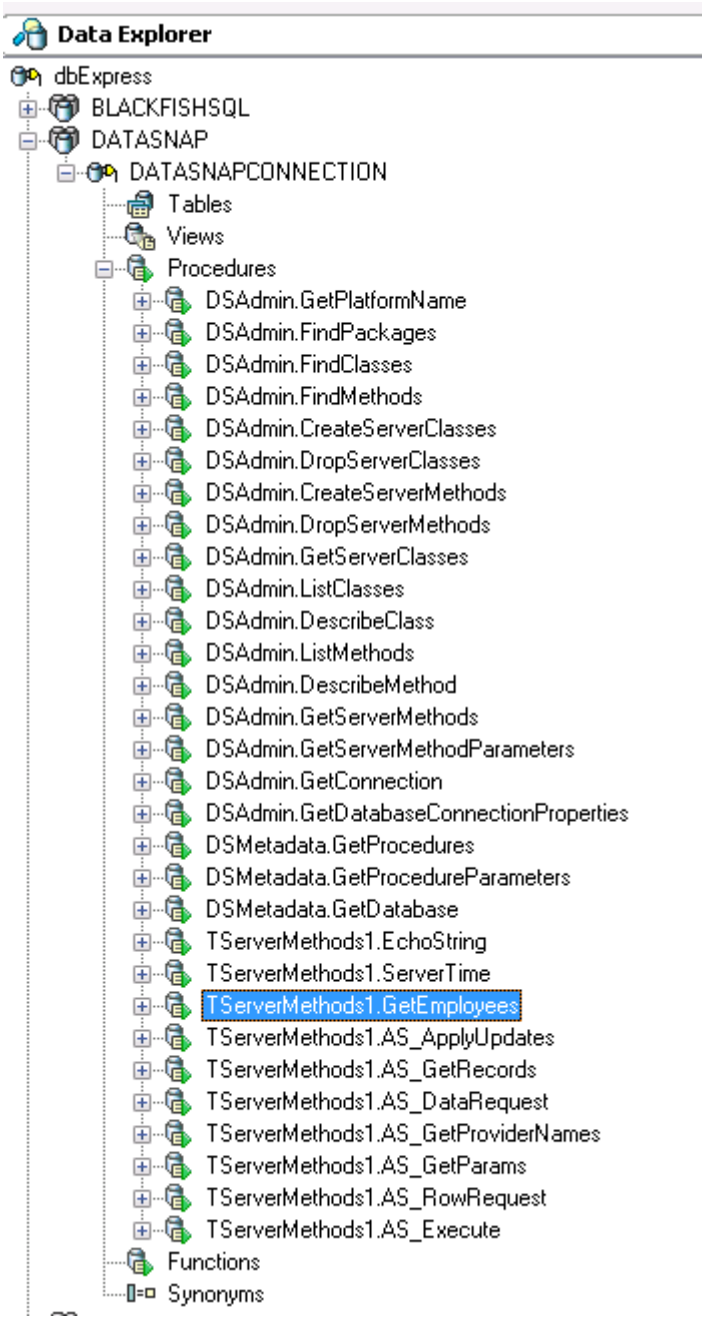
When adding functionality, we only have to work on the ServerMethodUnit1.pas unit, which is shared by both targets. By default, one sample method is included already, but like the Windows versions of the DataSnap Server, we can add two more methods (refer to section 2.1.4 for the components and source code needed on the Server Methods unit.
Once the server methods are implemented, we can deploy the ISAPI DLL on a web server like Microsoft's Internet Information Services. This is explained in detail in an article by Jim Tierney at http://blogs.embarcadero.com/jimtierney/2009/08/20/31502 so I won't repeat it again.

In case you do not have a web server available for deployment, you can play along with the DataSnap ISAPI Server as deployed on my server. Note that I have not exposed a TDataSetProvider, nor am I returning any data in the GetEmployees method, but the ServerTime and EchoString methods are working fine and should be enough to allow you to write a test DataSnap client against this server.

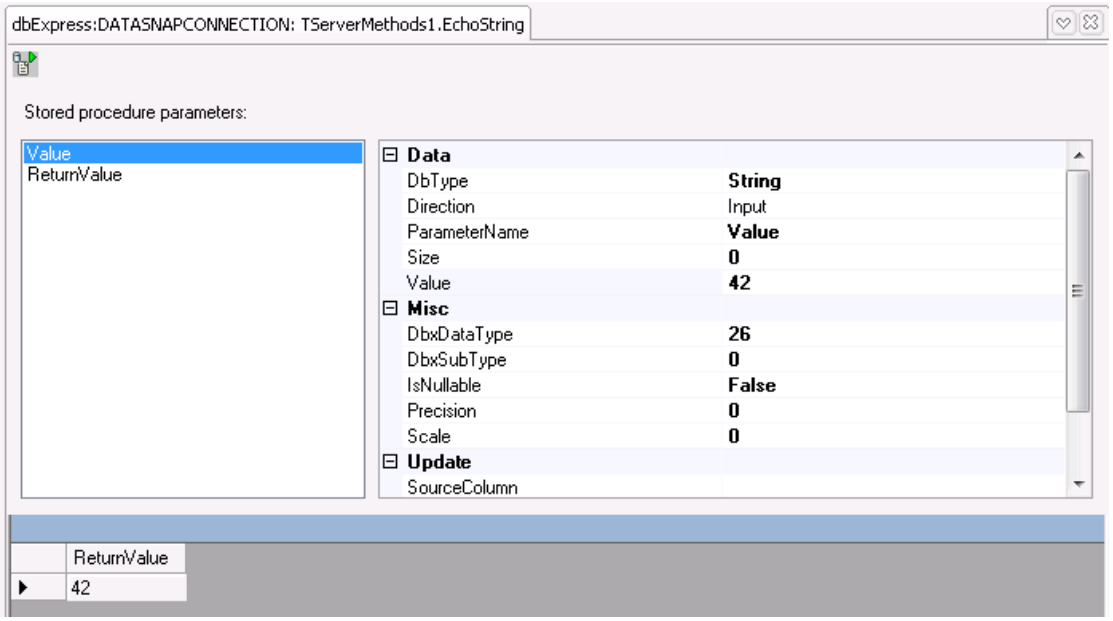
Before you want to connect to the ISAPI DataSnap Server inside a client application, it may be a good idea to use the Data Explorer to see if you can make a connection to the ISAPI DataSnap Server. The Data Explorer has a new Category called DATASNAP now, and if you open it up, there's a first connection available called DATASNAPCONNECTION that you can modify (just right-click on it and select Modify Connection).
In this dialog, we can specify the Protocol, Host (you can use www.bobswart.nl if you do not have your own web server available), Port, as well as the URL Path to the ISAPI DataSnap Server application on the server, which is cgi-bin/DSISAPIServer.dll. Click on Test Connection to make sure everything works.



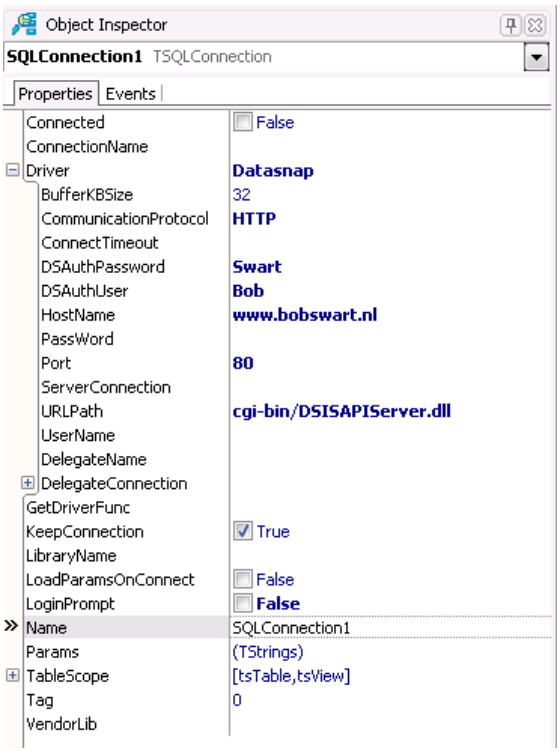
Now, click on OK to close the dialog, and in the Data Explorer, you can now expand the DATASNAPCONNECTION node to see the Tables, Views, Procedures, Functions and Synonyms. As you can see, the Procedures include all DSAdmin, DSMetaData, TServerMethods1.AS_xxx as well as our three custom server methods EchoString, ServerTime and GetEmployees.



Without the need to write a DataSnap client application, we can now test some of these methods. For example the EchoString (to ensure that what we send is also coming back). If we right-click on the TServerMethods1.GetEmployees procedure, we can select "View Parameters". This displays a new window in the IDE, where we can enter the value for the Value parameter (for example "42"). Then, we can right-click on this new window and select Execute to execute the remote server method. The result is shown as ReturnValue below:



This should prove that we can call the custom server methods from the DataSnap Server. In order to connect the DataSnap Client application to the server, we only have to modify the TSQLConnection properties. Previously, we connected to the Windows version of the DataSnap Server, but now we have to modify these settings to connect to the Web version instead.



Remember if you really want to use the DSISAPIServer.dll from my web server that I've disabled the TDataSetProvider and am not returning any data in the GetEmployees methods, but you can call the ServerTime and EchoString methods.

6. REST AND JSON – HOW YOU WANT IT

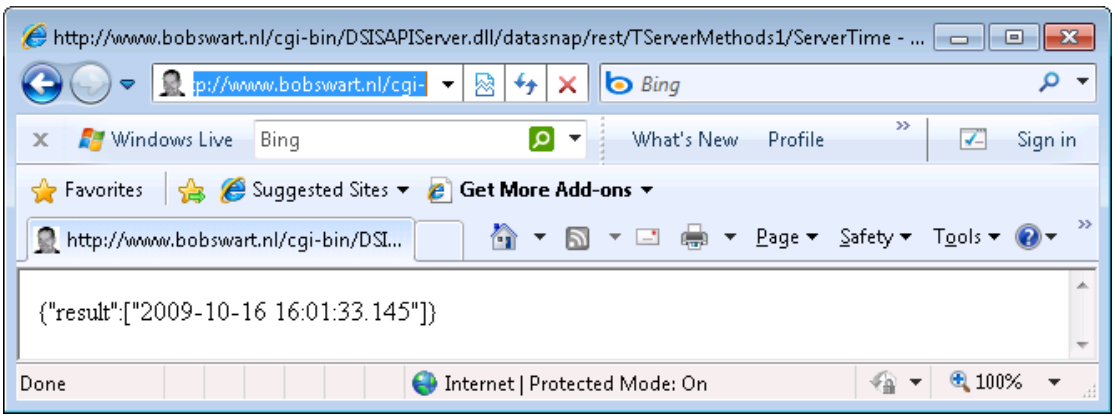
DataSnap 2010 supports both REST and JSON. DataSnap 2010 features REST support for DataSnap HTTP requests. For example, if the URL to the DataSnap Server is http://www.bobswart.nl/cgi-bin/DSISAPIServer.dll, then we can add /datasnap/rest to this URL, followed by the name of the Server Method class, the method, and the arguments. The generic syntax is as follows:

```
http://server/datasnap/rest/<class>/<method>/<parameters>
```

For the ServerTime method in the TServerMethod1 module of the DSISAPIServer.dll on my server, the URL is as follows:

```
http://www.bobswart.nl/cgi-bin/DSISAPIServer.dll/datasnap/rest/TServerMethods1/ServerTime
```

Calling this REST-enabled URL results in a JSON result, for example in the browser:



The result visible in the browser is a JSON construct:

```
{"result":["2009-10-16 16:01:33.145"]}
```

Marco Cantù will cover more REST details in his white paper on Delphi 2010 and REST clients.

6.1. CALLBACKS

Apart from being the result of REST-enabled calls to the DataSnap Servers, JSON is also used when implementing callback methods. DataSnap 2010 supports client-side callback functions, executed in the context of a server method. This means that during the execution of a server method (which is called by the DataSnap client), the server can call a callback function which was passed as argument to the server method by the client.

As an example, let's modify the EchoString method in order to add a callback function to it. The definition of the method EchoString should be modified as follows:

```
function EchoString(Value: string; callback: TDBXcallback): string;
```

The TDBXcallback type is defined in the DBXJSON unit. Before we can implement the new EchoString method, we should first see how the callback method can be defined at the client side (after all, it is a client method which can be called by the server).
At the client side, we must declare a new class, derived from TDBXCallback, and override the Execute method.

```
type
  TCallbackClient = class(TDBXCallback)
```

```
  public
    function Execute(const Arg: TJSONValue): TJSONValue; override;
  end;
```

Inside the Execute method, we get the Arg argument of type TJSONValue, which we can clone and then get our hands on the actual contents. The Execute method could also return a TJSONValue itself, so I'm just return the same value again.

```
function TCallbackClient.Execute(const Arg: TJSONValue): TJSONValue;
var
  Data: TJSONValue;
begin
  Data := TJSONValue(Arg.Clone);
  ShowMessage('Callback: ' + TJSONObject(Data).Get(0).JSonValue.value);
  Result := Data
end;
```

For this example, the callback method will show the value that was passed to the EchoString method, before the method actually returns (i.e. while the method is still being executed). The implementation of the new EchoString method at the server side should now put the string value inside a TJSONObject and pass it to the callback.Execute method, as follows:

```
function TServerMethods2.EchoString(Value: string; callback: TDBXcallback): string;
var
  msg: TJSONObject;
  pair: TJSONPair;
begin
  Result := Value;

  msg := TJSONObject.Create;
  pair := TJSONPair.Create('ECHO', Value);
  pair.Owned := True;
  msg.AddPair(pair);
  callback.Execute(msg);
end;
```

Note that the callback function is executed (at the client side) – and will return – before the actual EchoString method finished at the server side.
Finally, the call to the EchoString method at the client side also needs to change, since we now need to pass a callback class – an instance of our new TCallbackClient - as second argument.

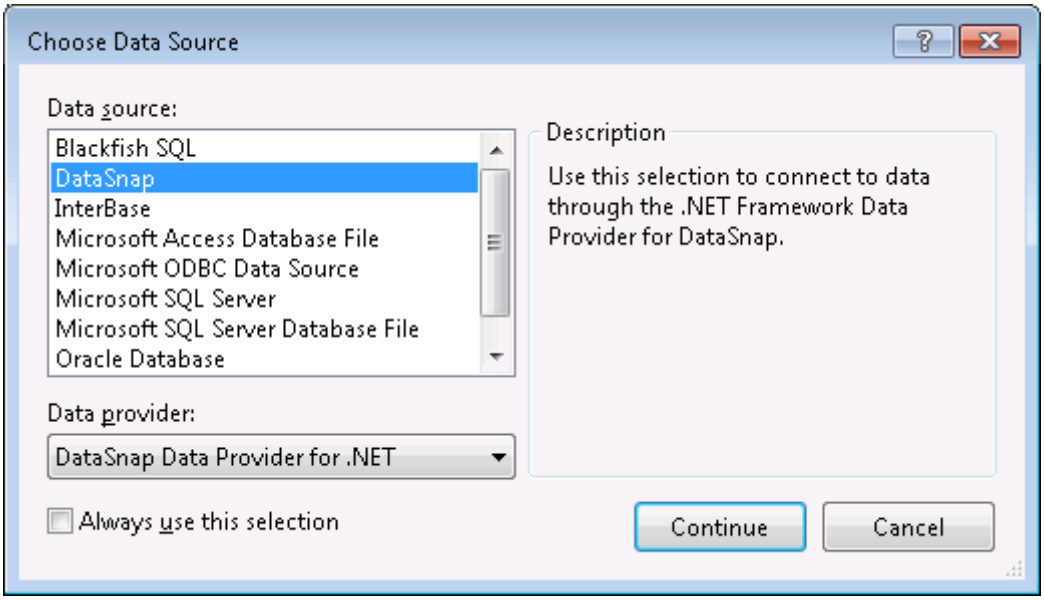
```
var
  MyCallback: TCallbackClient;
begin
  MyCallback := TCallbackClient.Create;
  try
    Server.EchoString(Edit1.text, MyCallback);
  finally
    MyCallback.Free;
  end;
end;
```

This simple example demonstrates how to use client-side callback methods in DataSnap 2010.

7. DataSnap and .NET – where you want it (more)

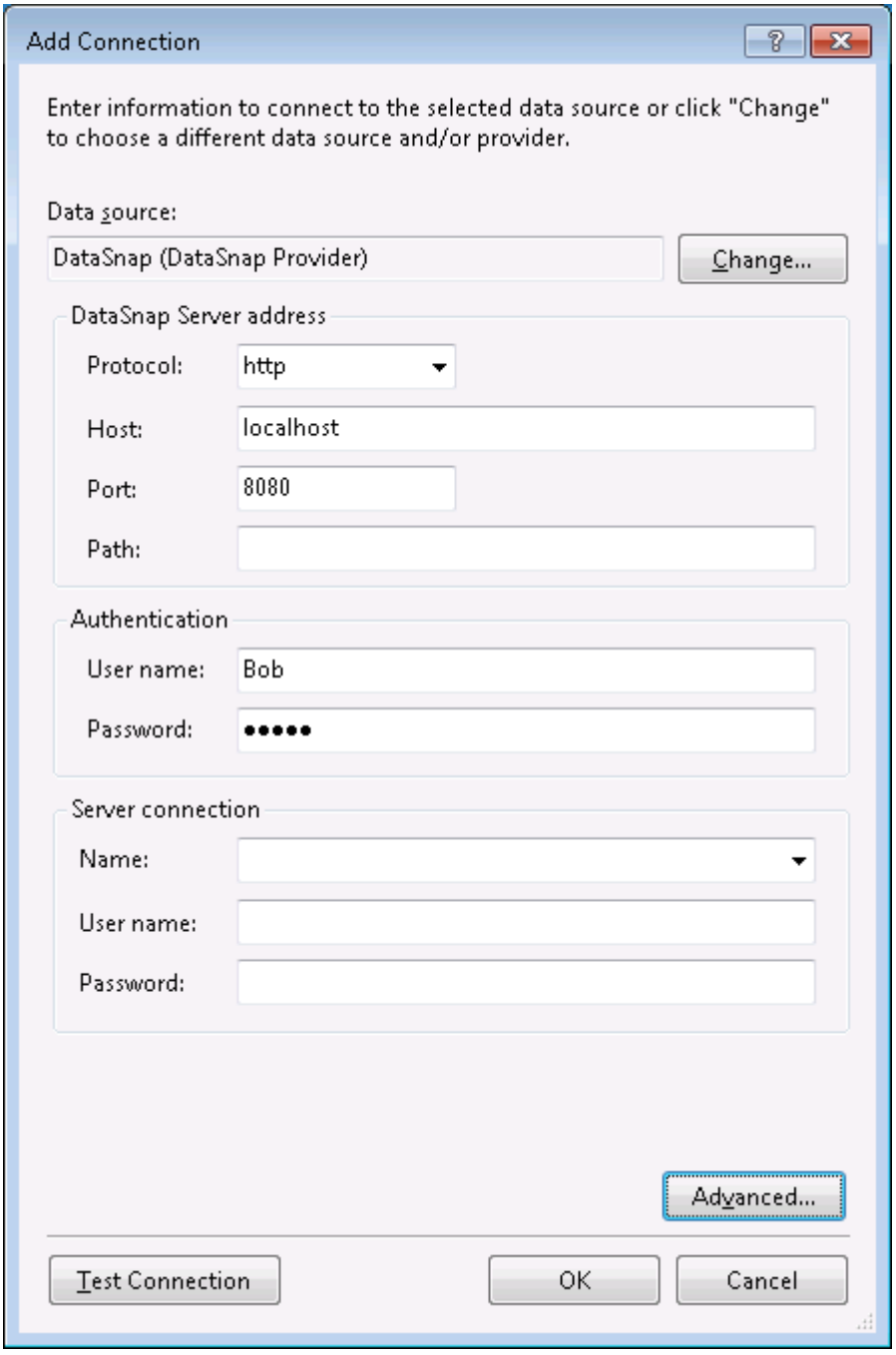
Delphi Prism 2010 is used to build a DataSnap .NET client for the Win32 servers we've made so far. In order to build the Delphi Prism 2010 DataSnap Client, make sure a DataSnap Server is running so we can connect to it during design-time already.

Start Delphi Prism 2010, and do View | Server Explorer to view the Delphi Prism Server Explorer. We should first make a connection here, to verify that we can actually work with the DataSnap Server.
The Server Explorer is a treeview with a root node called Data Connections. Right-click on Data Connections and select Add Connection. In the dialog that follows, select DataSnap from the list of data sources (note: you need to click on Change if a datasource is already preselected).



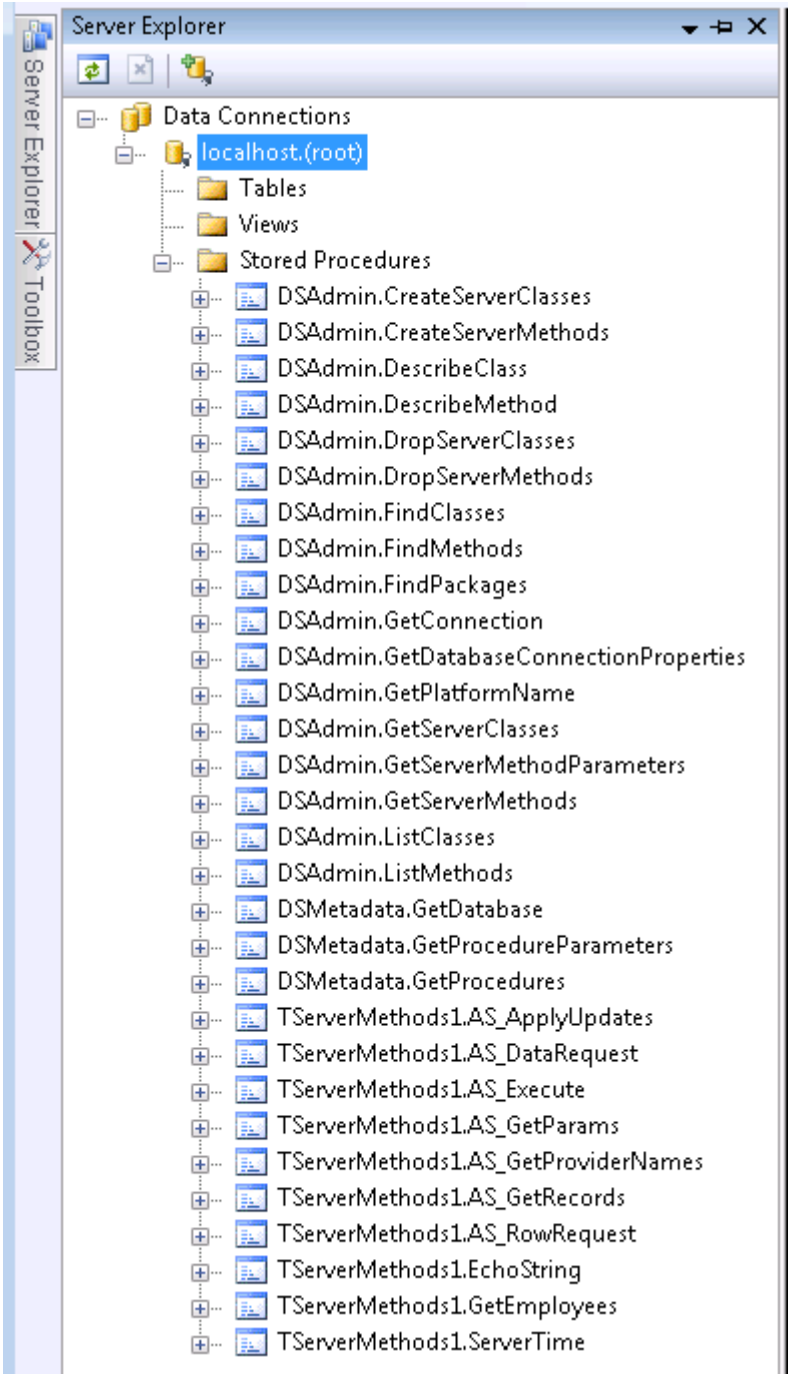
You may want to uncheck the "Always use this selection" checkbox, unless you always want to build only DataSnap data connections, of course.

Click on Continue to get to the next page of the dialog. Here, we can specify the details to connect to the DataSnap Server. In the Protocol drop-down combobox, we can select tcp/ip or http. Next, we should specify the Host (i.e. the machine name where the DataSnap Server is running – this can be localhost if you are testing on the same local machine). Then you need to specify the Port number. By default this will be Port 80 for HTTP and Post 211 for TCP/IP, but if you've read this white paper you will know that both values could (or should) be different – at least make sure to specify the same value here that you specified in the transport component(s) on the ServerContainerUnitDemo unit.
The next property contains the Path. This is only important if you want to connect to a Web Broker based DataSnap Server (where you need to specify the URLPath to get to the DataSnap Web Server – the part after the http://..../ domain part, that is).
Finally, don't forget to specify the Authentication User name and Password, in case the DataSnap Server is using HTTPAuthentication.

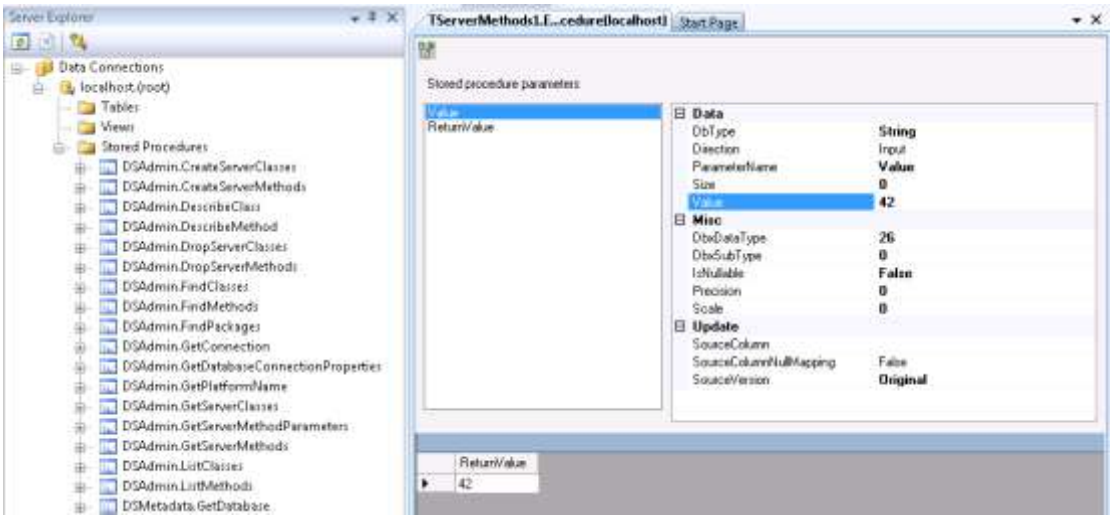


Click on the Test Connection button to verify that a connection can be made to the specified DataSnap Server. This will give you a "Test connection succeeded" dialog if everything was specified correctly.

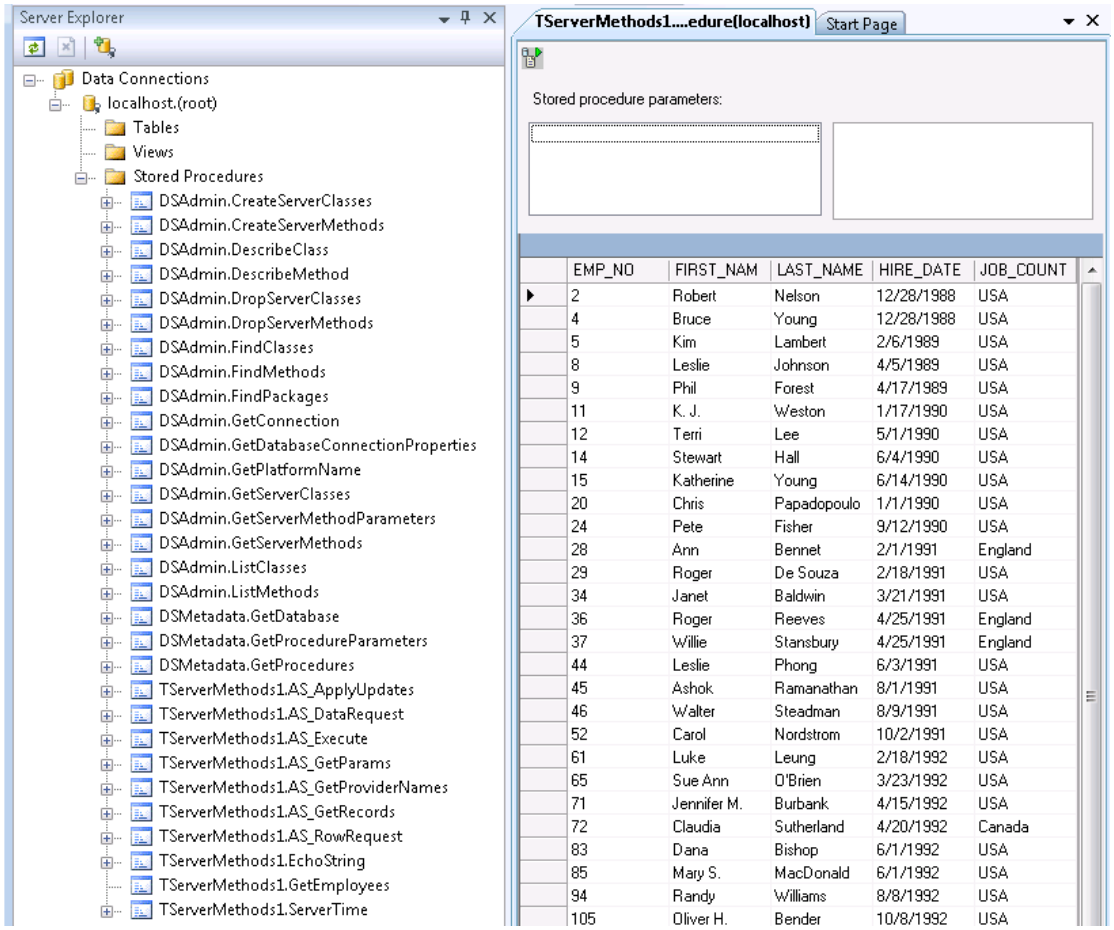
When you click on OK, a new entry for the DataSnap connection will be added to the Data Connections tree. In this case, it's for a localhost node. If you expand the new node, you'll find subnodes for Tables, Views and Stored Procedures. The Tables and Views are empty, but the Stored Procedures will contain all exposed Server Methods from the DataSnap Server. Including our custom server methods EchoString, GetEmployees and ServerTime.



We can now test some of the server methods from the Server Explorer. For example, right-click on the EchoString method and select View Parameters. This will give you a new window where you can enter a value for the Value parameter. Let's enter 42 as value for the Value parameter. Now, right-click in the window and select "Execute". This will execute the EchoString method from the DataSnap Server, showing the result just below the stored procedure parameters window:

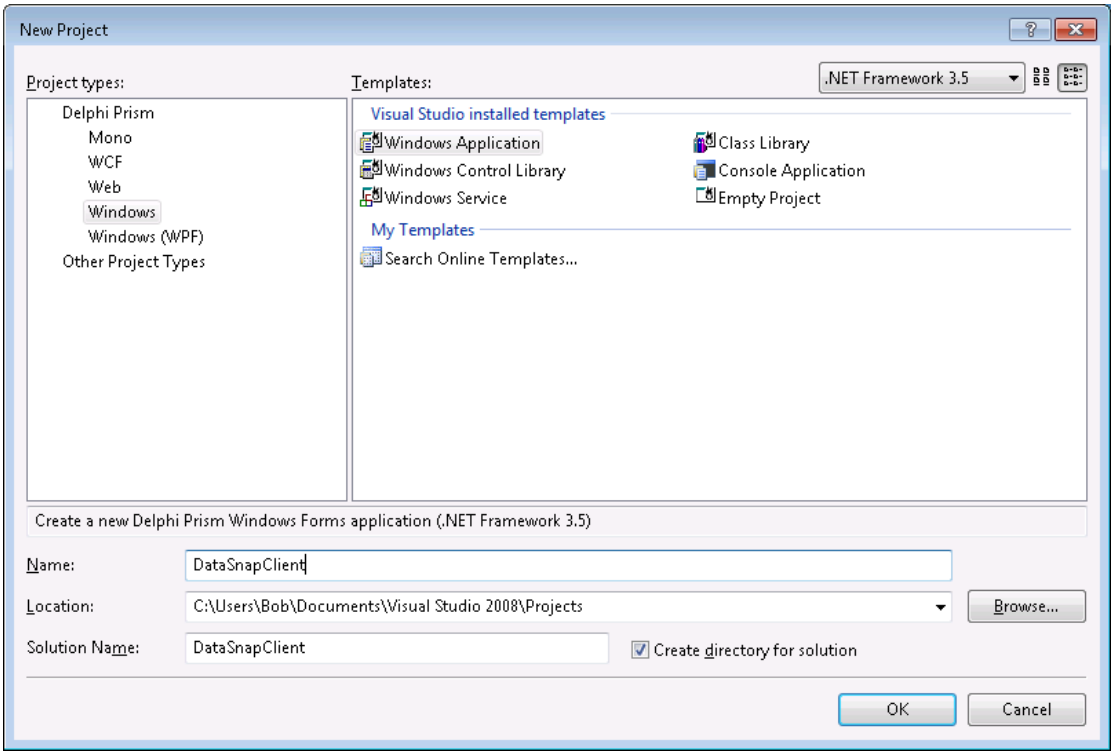


While this is nice, it's probably a bit more instructive to see how we can retrieve and use the data from the Employees table, by using the GetEmployees method. This Stored Procedure has no parameters, but we can still select the "View Parameters" command, which just gives an empty list of stored procedure parameters. Again, right-click on this window and select "Execute". This time, the result is the complete set of records from the Employees table, as returned by the GetEmployees method:



7.1. WinForms Client

Although working with DataSnap Server methods in the Server Explorer can be fun, it's more useful to call them from a .NET application. For the last example, do File | New Project to start the New Project wizard in Delphi Prism. This will give you an overview of the available targets.
From the Windows Project Type, select the Windows Application and change the Name from WindowsApplication1 to DataSnapClient.



If you click on OK, a new project DataSnapClient will be created in the Delphi Prism IDE, with a Main.pas unit for the Main Form.

From Server Explorer, select the new connection to the DataSnap Server we created in the previous section. The Properties Explorer will display the properties, including the ConnectionString, which should be something like the following:

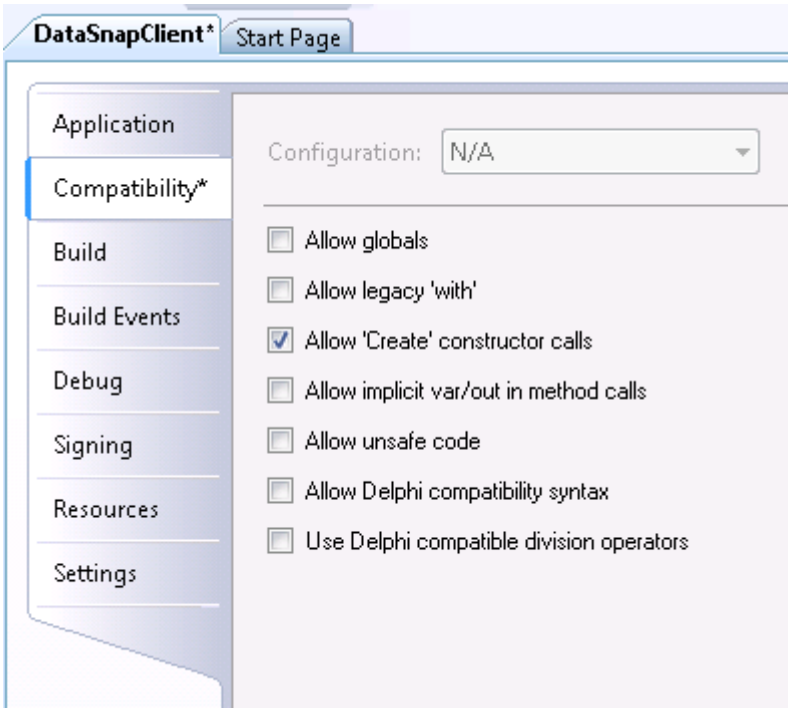
```
communicationprotocol=http;hostname=localhost;port=8080;dsauthenticationuser=Bob;ds
authenticationpassword=Swart
```

Right-click on the data connection node, and select "Generate Client Proxy" option. This will generate a file ClientProxy1.pas, with the definition of a class called TServerMethods1Client with a number of method, including EchoString, ServerTime, and GetEmployees. A snippet from the class definition is as follows:

```
TServerMethods1Client = class
public
  constructor (ADBXConnection: TAdoDbxConnection);
  constructor (ADBXConnection: TAdoDbxConnection; AInstanceOwner: Boolean);
  function EchoString(Value: string): string;
  function ServerTime: DateTime;
  function GetEmployees: System.Data.IDataReader;
```

Apart from the proxy class, there are also a number of references added to the References node of the project: Borland.Data.AdoDbxClient and Borland.Data.DbxClientDriver to be precise.

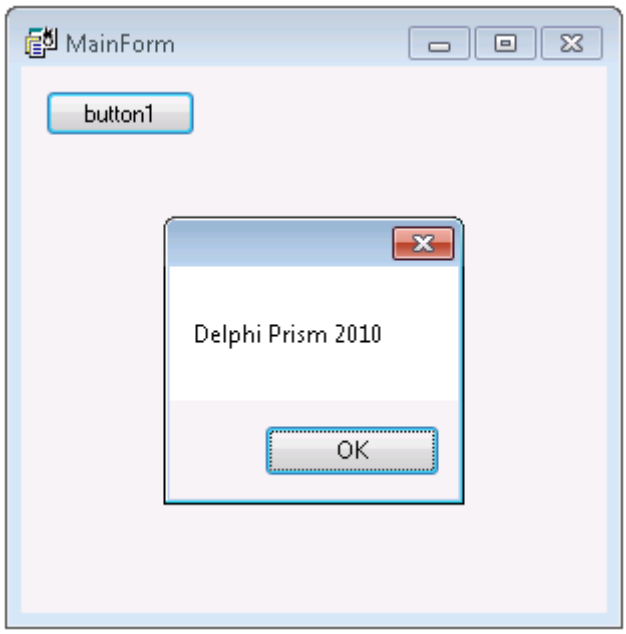
As you can see from the code snippet of the TServerMethods1Client, this class has two constructors: both with a ADBXConnection parameter, and the second one also with an AInstanceOwner Boolean parameter. This means we need to call the constructor with an argument. And in order to support that, we have to make a modification to the project settings. Right-click on the DataSnapClient node in the Solution Explorer, and select Properties. In the window that is now shown, click on the Compatibility tab and then check the option "Allow Create constructor calls", which will allow us to call the .Create constructor, passing arguments, instead of using the new operator.



Now, we can return to the Main Form, and place a Button on it. In the Click event of the button, we can create a connection to the DataSnap Server and call one of its methods.

```
method MainForm.button1_Click(sender: System.Object; e: System.EventArgs);
var
  Client: ClientProxy1.TServerMethods1Client;
  Connection: Borland.Data.TAdoDbxDatasnapConnection;
begin
  Connection := new Borland.Data.TAdoDbxDatasnapConnection();
  Connection.ConnectionString :=

'communicationprotocol=http;hostname=localhost;port=8080;dsauthenticationuser=Bob;dsau
enticationpassword=Swart';
  Connection.Open;
  try
    Client := ClientProxy1.TServerMethods1Client.Create(Connection);
    MessageBox.Show(
      Client.EchoString('Delphi Prism 2010'));
  finally
    Connection.Close;
  end;
end;
```

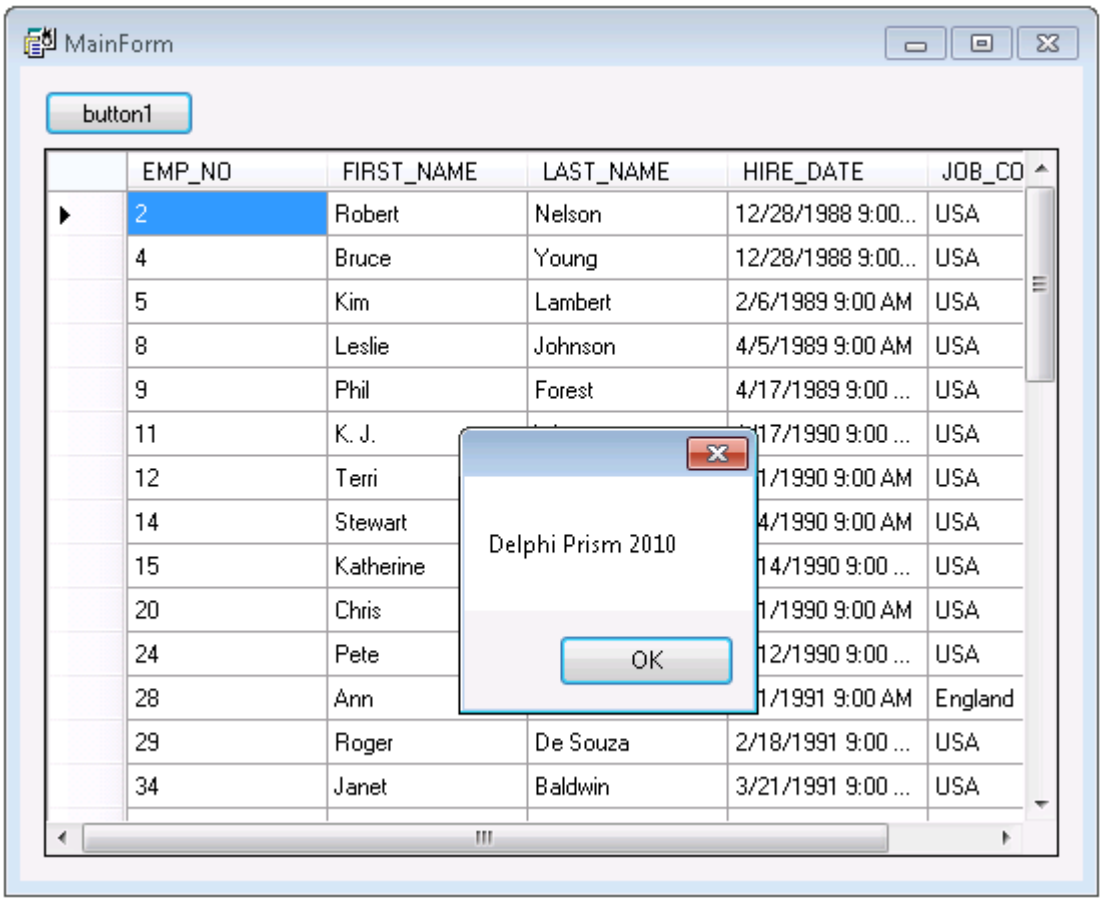
The result is the echo of Delphi Prism 2010, as can be seen below.



In a similar way, we can call the GetEmployees method and assign the result to a DataGridView. This poses us with a little problem, since the GetEmployees will return an IDataReader (the equivalent of a TSQLDataSet's result), and not a DataSet or a DataTable. We have to write a few lines of code to load the result of GetEmployees into a new DataTable inside a DataSet (the equivalent of the TClientDataSet at the Win32 side).

```
method MainForm.button1_Click(sender: System.Object; e: System.EventArgs);
var
  Client: ClientProxy1.TServerMethods1Client;
  Connection: Borland.Data.TAdoDbxDatasnapConnection;
  Employees: System.Data.IDataReader;
  ds: System.Data.DataSet;
  dt: System.Data.DataTable;
begin
  Connection := new Borland.Data.TAdoDbxDatasnapConnection();
  Connection.ConnectionString :=

'communicationprotocol=http;hostname=localhost;port=8080;dsauthenticationuser=Bob;dsau
enticationpassword=Swart';
  Connection.Open;
  try
    Client := ClientProxy1.TServerMethods1Client.Create(Connection);
    Employees := Client.GetEmployees;
    ds := new DataSet();
    dt := new DataTable("DataSnap");
    ds.Tables.Add(dt);
    ds.Load(Employees, LoadOPtion.PreserveChanges, ds.Tables[0]);
    dataGridView1.DataSource := ds.Tables[0];

    MessageBox.Show(
      Client.EchoString('Delphi Prism 2010'));
  finally
    Connection.Close;
  end;
end;
```

The result is the following data shown in the DataGridView of a Delphi Prism WinForms application, which demonstrates that we can write thin .NET clients to connect to DataSnap Servers.



8. SUMMARY

In this white paper, I've explained that you can use DataSnap where you want (on Windows: with a GUI, service or console application, or on the web with a CGI, ISAPI or Web App Debugger application), as well as Win32 or .NET clients, and how you want – using TCP/IP, HTTP, with HTTP Authentication and optionally some filters for compression, encryption, etc. DataSnap 2010 is substantially expanded and enhanced compared to DataSnap 2009, and a lot improved since the COM-based original versions of DataSnap and MIDAS.

Bob Swart – Bob Swart Training & Consultancy (eBob4)^

Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster and run them better, regardless of their platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs, simplify change management and compliance and accelerate innovation. The company's flagship tools include: Embarcadero® Change Manager™, Embarcadero™ RAD Studio, DBArtisan®, Delphi®, ER/Studio®, JBuilder® and Rapid SQL®. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. Embarcadero is online at www.embarcadero.com..