Study Guide

embarcadero

GETTING STARTED WITH WINDOWS AND MAC DEVELOPMENT

Building

Multi-Client, Multi-Platform, Multi-Tier

Applications

E-Learning Series Course Book

Lesson 9

Embarcadero Technologies

Americas 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

Lesson 9 – Building Multi-Client, Multi-Platform, Multi-Tier Applications

Version: 0.99
Presented: June 28, 2012
Last Updated: August 1, 2012
Prepared by: David Intersimone "David I", Embarcadero Technologies

davidi@embarcadero.com
http://blogs.embarcadero.com/davidi/

Contents

Introduction

Not every application is a standalone desktop application that runs on just one platform. Software systems often involve a series of client applications working with remote databases and remote business objects. There will be many times when you will need to build multi-client, multi-platform, multi-tier applications for Windows and Mac. As you have seen in the earlier lessons, you can create multi-platform Windows, Mac and iOS applications that have HD and 3D user interfaces and connect to Data. What if you need to build and/or connect to business objects and services as part of your architecture? RAD Studio supports the building of multi-client, multi-platform and multi-tier applications.

RAD Studio includes the following technologies and components that support the creation of applications for multi-client, multi-platform and multi-tier:

- FireMonkey HD and 3D client applications – Win32 (Delphi and C++), Win64 (Delphi), Mac OSX (Delphi and C++) client applications
- VCL client applications - Win32 (Delphi and C++), Win64 (Delphi)
- Console client applications – Win32 (Delphi and C++), Win64 (Delphi), Mac OSX (Delphi and C++)
- Web service client access – Win32 (Delphi and C++), Win64 (Delphi), Mac OSX (Delphi and C++)
- Cloud storage client application access – Amazon S3 and Microsoft Windows Azure
- DataSnap client application access – Win32 (Delphi and C++), Win64 (Delphi), Mac OSX (Delphi and C++)
- DataSnap Application Servers – VCL, FireMonkey, Console Application, Service Application - Win32 (Delphi and C++), Win64 (Delphi)
- Web Application Servers – Win32 (Delphi and C++), Win64 (Delphi)
- Web Services - Win32 (Delphi and C++) and Win64 (Delphi)

In lesson 9 we'll focus on building multi-client, multi-platform, multi-tier applications using FireMonkey, DataSnap, Web Services and Cloud Storage.

Creating Multi-Tier Applications

A multi-tiered application is partitioned into logical units, called tiers, which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many advantages of the multi-tiered database model, such as centralized business logic and thin client applications.

In its simplest form, sometimes called the "three-tiered model", a multi-tiered application is partitioned into thirds:

- Client application: provides a user interface on the user's machine.
- Application server: resides in a central networking location accessible to all clients and provides common data services.
- Remote database server: provides the relational database management system (RDBMS).

In this three-tiered model, the application server manages the flow of data between clients and the remote database server, so it is sometimes called a "data broker." You usually only create the application server and its clients, although, if you are really ambitious, you could create your own database back end as well.

In more complex multi-tiered applications, additional services reside between a client and a remote database server. For example, there might be a security services broker to handle secure Internet transactions, or bridge services to handle sharing of data with databases on other platforms.

Support for developing multi-tiered applications is an extension of the way client datasets communicate with a provider component using transportable data packets. Once you understand how to create and manage a three-tiered application, you can create and add additional service layers based on your needs.

The advantages of this multi-tiered model include the following:

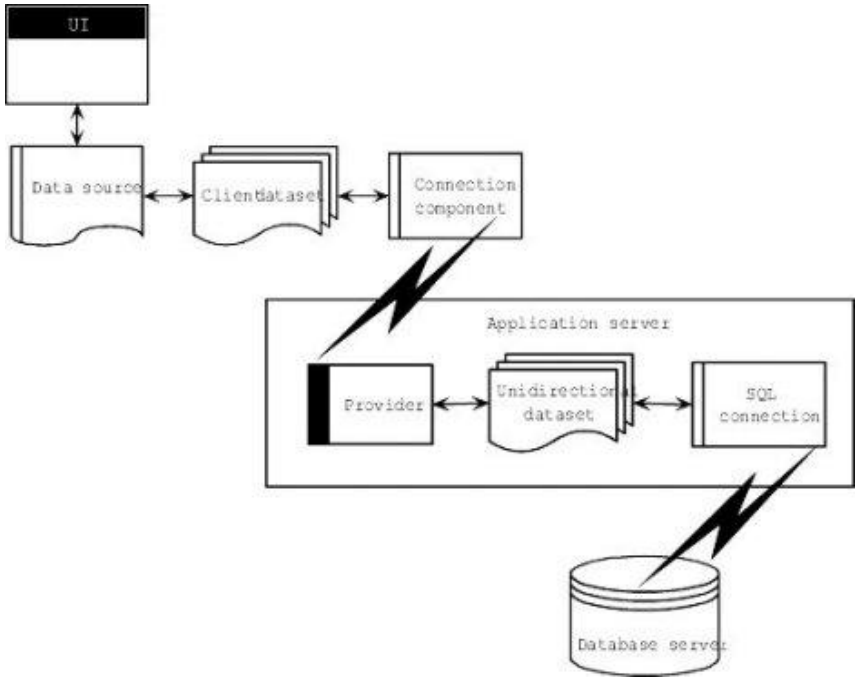
- Encapsulation of business logic in a shared middle tier. Different client applications all access the same middle tier. This allows you to avoid the redundancy (and maintenance cost) of duplicating your business rules for each separate client application.

- Thin client applications. Your client applications can be written to make a small footprint by delegating more of the processing to middle tiers. Not only are client applications smaller, but they are easier to deploy because they don't need to worry about installing, configuring, and maintaining the database connectivity software (such as the database server's client-side software). Thin client applications can be distributed over the Internet for additional flexibility.
- Distributed data processing. Distributing the work of an application over several machines can improve performance because of load balancing, and allow redundant systems to take over when a server goes down.
- Increased opportunity for security. You can isolate sensitive functionality into tiers that have different access restrictions. This provides flexible and configurable levels of security. Middle tiers can limit the entry points to sensitive material, allowing you to control access more easily. If you are using HTTPS, you can take advantage of the security models it supports.

Multi-tiered applications use the components on the DataSnap page, the Data Access page, and possibly the Web Services page of the Tool Palette, plus a remote data module that is created by a wizard on the Multitier or Web Services page of the New Items dialog. They are based on the ability of provider components to package data into transportable data packets and handle updates received as transportable delta packets.

The components needed for a multi-tiered application include:

- Remote and Server data modules - Specialized data modules that can act as a COM Automation server, a DataSnap data module, RESTful web servers, or implement a Web Service to give client applications access to any providers they contain. Used on the application server.
- Provider component - A data broker that provides data by creating data packets and resolves client updates. Used on the application server.
- Client dataset component - A specialized dataset used to manage data stored in data packets. The client dataset is used in the client application. It caches updates locally, and applies them in delta packets to the application server.
- Connection components - A family of components that locate the server, form connections, and make the IAppServer interface available to client datasets. Each connection component is specialized to use a particular communications protocol.



The provider and client dataset components require midas.dll or midaslib.dcu, which manages datasets stored as data packets. Note that, because the provider is used on the application server and the client dataset is used on the client application, if you are using midas.dll, you must deploy it on both application server and client application. See the deploy.htm file in your installation directory for additional important information about redistributables.

RAD Studio Target Platforms

RAD Studio supports the following target platforms and associated frameworks:

- Mac OS X hosts only FireMonkey development (Delphi and C++Builder).
- 32-bit Windows hosts FireMonkey and VCL development (Delphi and C++Builder).
- 64-bit Windows hosts FireMonkey and VCL development (Delphi only).

You specify the target platform for your application by selecting from the Targets node of the Project Manager.

Other Multi-Platform Development support is included for

- FireMonkey application development for iOS devices is supported for iPhone, iPad, and iPod. However, FMX iOS is not technically a target platform that you can select in the IDE.
- DataSnap supports native-language development for mobile devices using special DataSnap connectors that we provide for (using the mobile connectors is beyond the scope of this getting started course – see links to articles and videos at the end of this lesson if you need to support these mobile devices):
    - iOS devices using the Objective-C iOS DataSnap proxy
    - iOS devices using the Free Pascal DataSnap proxy
    - Windows 7 Phone using the C# Silverlight DataSnap proxy
    - Android using the Java Android DataSnap proxy
    - Blackberry using the Java BlackBerry DataSnap proxy

The Embarcadero DocWiki has a summary of the types of multi-platform applications you can build at http://docwiki.embarcadero.com/RADStudio/en/Types_of_Cross-Platform_Applications_You_Can_Create

For information about building Web Applications using WebSnap, read the Embarcadero DocWiki article at http://docwiki.embarcadero.com/RADStudio/en/Developing_Web_Applications_with_WebSnap.

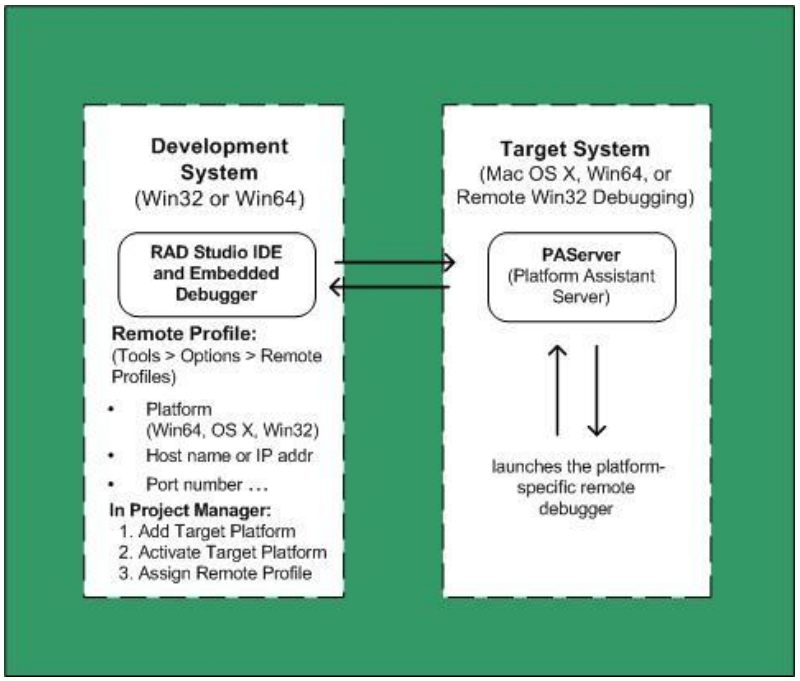
Debugging Multi-Client, Multi-Platform, Multi-Tier Applications

The integrated RAD Studio debugger enables you to debug an application that targets any of the Supported Target Platforms. The integrated debugger works with both Delphi and C++ applications.

RAD Studio supports two cross-platform debuggers:

- Embarcadero Win64 Debugger (on a PC running a 64-bit Windows OS)
- Embarcadero OS X Debugger (on a Mac running a version of OS X)

When you install the Platform Assistant on your target platform, the cross-platform debuggers are are also installed. The debuggers that are registered on your system are listed in **Tools > Options > Debugger Options**.



The Platform Assistant and a remote profile are required for establishing a debug session for an application running on a cross-platform target system. The only cross-platform configuration that does not require the Platform Assistant is a Win64 development system.

The cross-platform debugger and the process to be debugged both run on the target platform. The cross-platform debugger reports status and interacts with you on the development PC in the RAD Studio IDE. This means that using one of the cross-platform debuggers is very similar to using the integrated debugger for Win32 applications.

To do cross-platform debugging:

- Your application must have an activated target platform (Win64, OS X, or remote Win32).
- Depending on the target platform:
    - For the Mac OS X:
        - The Platform Assistant must be running on the Mac.
        - Your application must have an assigned remote profile.
        - Your development system must have a live connection to the Mac (that is, Test Connection must succeed on the Remote Profiles window). This enables you to use both the integrated debugger and the Deployment Manager.
    - For 64-bit Windows:
        - If you are connecting to a remote 64-bit PC:
            - You must use the Platform Assistant and a remote profile.
            - Your development system must have a live connection to the remote 64-bit PC (that is, Test Connection must succeed on the Remote Profiles window). This enables you to use both the integrated debugger and the Deployment Manager.
        - If your development system is a PC running a 64-bit Windows operating system:

- You do not need to use the Platform Assistant because your debug environment is in-machine, and the integrated debugger runs automatically.
- You can, however, optionally choose to use the Platform Assistant and a remote profile for the target platform just as you would for a remote 64-bit Windows target system. Doing this enables you to use the Deployment Manager.

Debugging on OS X is a privileged operation; only a process with adequate access rights can act as a debugger. In order to guarantee support for debugging on the Mac, you need to log in on the Developer Tools Access dialog box using the administrator or root user password. The first time you start the Platform Assistant in a session (or when you start a debugging session on the Mac), the Mac displays a login dialog for Developer Tools Access. You need to provide the password for the root user on that Mac. The password is required only once per session.

Developing DataSnap Applications

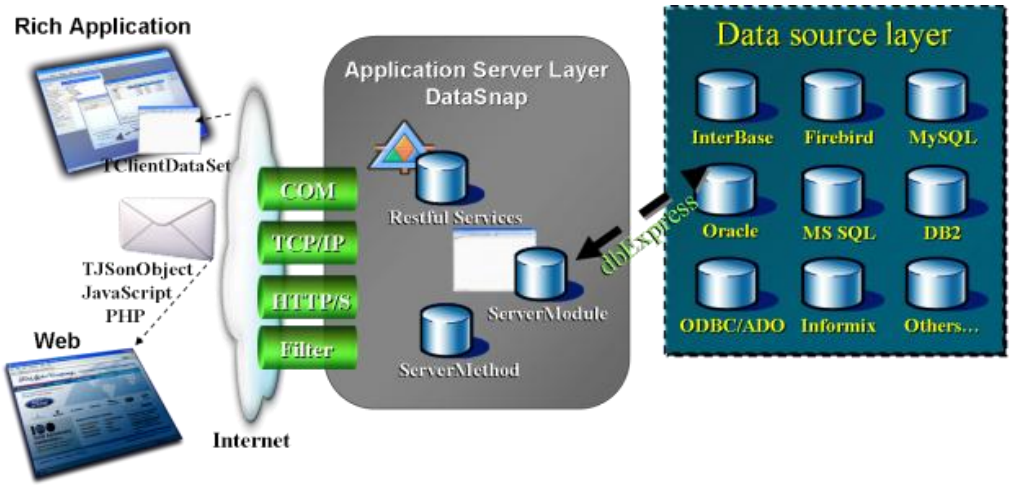
The DataSnap technology continues to evolve, as the demands for distributed computing increase. The technology behind DataSnap has moved beyond the approach of remoting data through the Microsoft COM/DCOM to a more open communication approach based on TCP/IP. This evolution has allowed the DataSnap technology to expand its capabilities in order to include a complete middleware technology. One of the key features of the technology is that it is fast: fast to build, fast to deploy, and fast to execute in production.

DataSnap now has expanded capabilities, allowing the technology to work within almost any standards-based infrastructure. While the latest DataSnap is still backwards-compatible with the COM/DCOM approach, it now has the ability to communicate natively through TCP/IP, and alternatively through HTTP or HTTPS. At the same time, the business logic found in the DataSnap servers can be broadcast as RESTful (Representational State Transfer) web services.

DataSnap provides a way for the Client to safely communicate with the Server, using a secured transfer of JSON (JavaScript Object Notation) data content over TCP/IP or HTTP. The ability to define filters at both ends of the communication channel, for encryption and compression purposes, improves the security.

Another benefit of the DataSnap technology is that it offers the possibility to asynchronously notify all the Client applications about changes made to the Server, so that Clients can take appropriate actions. The callback does not require Clients to invoke any of the Server methods.

Once you purchase the enterprise or architect edition of RAD Studio XE2, Delphi XE2 or C++Builder XE2, there are no additional charges to use or deploy the DataSnap technology.



You can find DataSnap sample applications (that are installed with RAD Studio) by choosing Start | Programs | Embarcadero RAD Studio XE2 | Samples. The folders of interest are DataSnapXE, containing DataSnap XE multiplatform demos, and Delphi\DataSnap, containing several DataSnap application examples. These DataSnap samples are called, by their folder name:

- Basic DataSnap Client and Server Sample - http://docwiki.embarcadero.com/CodeExamples/XE2/en/DataSnap.Basic_DataSnap_Client_and_Server_Sample
- Role Authorization Sample - http://docwiki.embarcadero.com/CodeExamples/XE2/en/DataSnap.Role_Authorization_Sample
- Chat Room Sample - http://docwiki.embarcadero.com/CodeExamples/XE2/en/DataSnap.ChatRoom_Sample
- Failover - DataSnap HTTP Tunneling Sample - http://docwiki.embarcadero.com/CodeExamples/XE2/en/DataSnap.Failover_-_DataSnap_HTTP_Tunneling_Sample
- JSON Viewer Sample - http://docwiki.embarcadero.com/CodeExamples/XE2/en/DataSnapJSON_Viewer_Sample
- Proxy Generator Sample - http://docwiki.embarcadero.com/CodeExamples/XE2/en/DataSnap.Proxy_Generator_Sample

Developing your first Windows and Mac DataSnap Application

DataSnap technology provides the ability to create Client-Server applications that communicate through the Internet, the local network, or the local host.

The following steps show you how to create and use DataSnap to build a simple server and client application. After creating the server and activating the connection between the client and server with DataSnap, the client can call methods defined and implemented on the DataSnap server. You'll learn how to use databases in your DataSnap applications in the next section.

You can implement a server in either Delphi or C++. This example shows both. The client does not need to be implemented in the same language. DataSnap allows you to have a Delphi server and a C++ client—or vice versa.

The main DataSnap Server Components are

- TDSServer - The TDSServer component is the logical heart of the DataSnap server application. It contains Start and Stop methods for starting and stopping the server. It also contains the AutoStart property. By default, the value of AutoStart is set to True, so the server starts automatically when the application does. You need only one TDSServer component per server application.

- TDSServerClass - The TDSServerClass component represents a server class.

The DataSnap server automatically creates and destroys instances of server classes. The instancing of a server class is controlled by the LifeCycle property of the TDSServerClass component. The LifeCycle property has three possible values: Server, Session, and Invocation.

- LifeCycle set to Server means that the DataSnap server creates one instance of a server class that is used by all clients connected to the server application. This represents a "singleton" pattern. Be careful when using the Server life cycle, because your server class implementation needs to be thread-safe: you must design the server class so that it can be accessed simultaneously from multiple threads.
- The default value of LifeCycle is Session. This means that the DataSnap server creates one instance of a server class for every connected client.
- The third possible value for the LifeCycle property is Invocation. In this case, a server class instance is created and destroyed for every method call arriving from a client, and the state of a server class is not preserved between method calls.

A component is also needed to provide communication between the client and server.

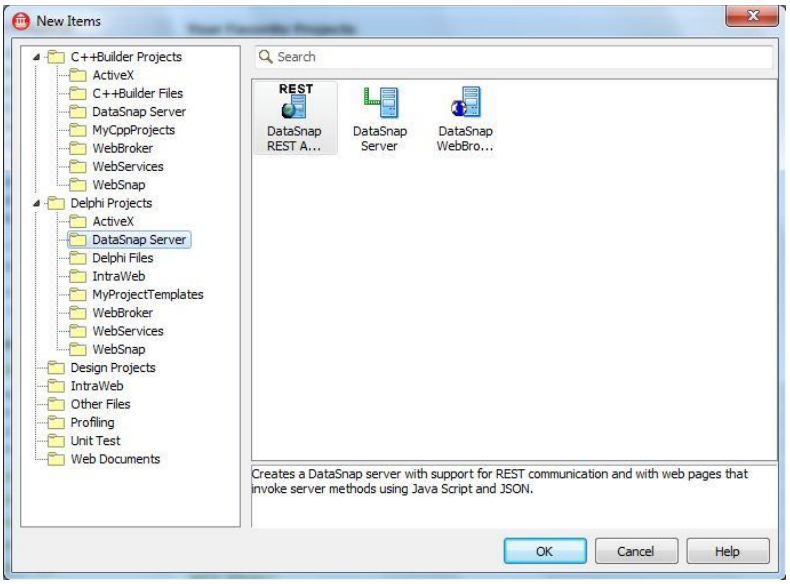
The TDSTCPServerTransport component implements a multithreaded TCP server listening for incoming client connections on multiple threads. This component does not have any events. The Port property indicates the TCP port to be used. By default, it is set to 211. You can also use HTTP for communication between the client and server.

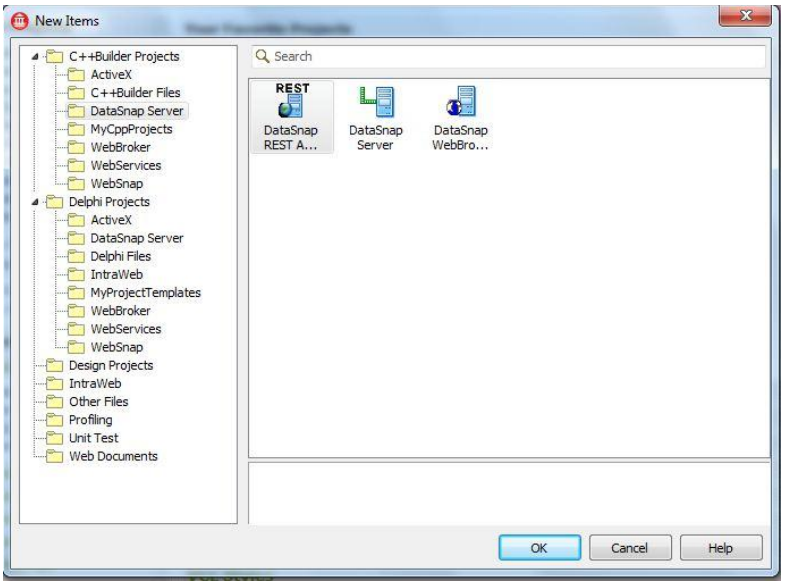
Step 1: Creating the DataSnap Server Application

Three wizards are included to help you start DataSnap projects for Delphi and C++:

- DataSnap Server – The DataSnap Server wizard provides an easy way to implement a server application using the DataSnap technology.
- DataSnap WebBroker Application – The DataSnap WebBroker Application wizard provides an easy way to implement a server application using both The WebBroker and DataSnap technologies.

- DataSnap REST Application – Creates a DataSnap Server with support for REST communication and with Web pages that invoke server methods using JavaScript and JSON.





You can also build a console application or FireMonkey application and add the DataSnap server components into these applications.

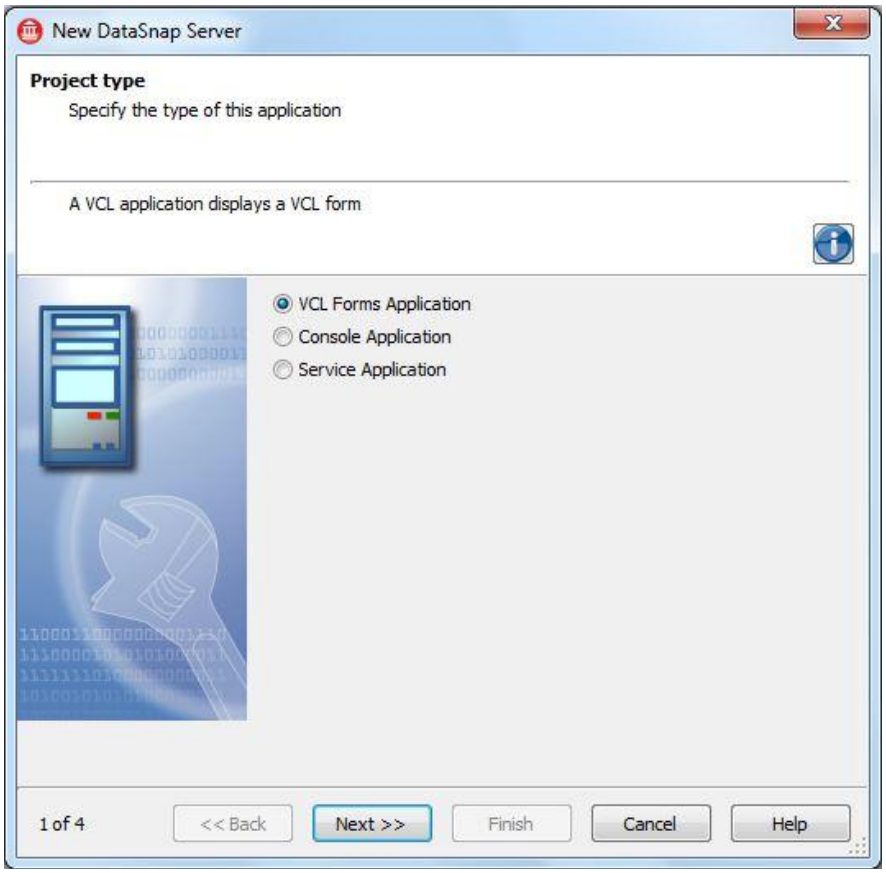
For this getting started lesson, we'll use the DataSnap Server wizard. To start the wizard use **File > New > Other…** and choose the DataSnap Server wizard from the Delphi Projects > DataSnap Server category or C++Builder Projects > DataSnap Server category.

After you select the Wizard you will see the wizard present 4 dialogs that will help you create your DataSnap Server application. The DataSnap Server Wizard provides an easy way to implement a server application using the DataSnap technology. Regardless of the options you select, it creates a ServerContainerUnit that contains a TDSServer component. All the additional components are

connected to the TDSServer, having the Server property set to the name of the TDSServer component. If the selected protocol is TCP/IP, the server unit also contains a TDSTCPServerTransport. If HTTP is selected as the communication protocol, the server unit contains a TDSHTTPService component. If HTTPS is selected, then the server unit will contain a TDSHTTPService component configured for HTTPS operations.

In the first step (step 1 of 4) the wizard gives you the choice for the type of application you want for your DataSnap server. Choices are: VCL application, Console Application and Service application. As I mentioned earlier, if you want your DataSnap Server application to have a UI you can create a FireMonkey project and put the DataSnap server components in your application.

To keep things simple, we'll create a VCL Forms Application for our DataSnap Server. This will give your DataSnap server a simple UI where you can place additional controls to display, for example, a dashboard of how many DataSnap clients are connected to the server.



The wizard next (step 2 of 4) asks you what server features you want included in your DataSnap Server application. Every feature in the check list is self-descriptive: select one and a hint will appear in the upper side of the wizard.

The available communication protocols are:

- TCP/IP
- HTTP

- HTTPS

You can select them in each combination you want, but make sure TCP/IP is always selected. Selecting a communication protocol from this list enables the selection of communication ports for the protocols that are selected, in the next step of the DataSnap Server Wizard. If you select HTTPS as the communication protocol, the DataSnap Server Wizard will display an additional page, asking for information regarding certificate files.

If you select the Authentication option, a TDSHTTPServiceAuthenticationManager component is placed on the server form. The TDSHTTPService component uses TDSHTTPServiceAuthenticationManager as the AuthenticationManager to allow the implementation of HTTP user authentication for the DataSnap server. The implementation consists in implementing the Authenticate event. When Authentication is selected, the client must provide the DataSnap user name and password as SQL connection properties.

Selecting the Server Methods Class option will add a TDSServerClass component to the server form that allows defining a class on the server, which will expose server methods to client applications.

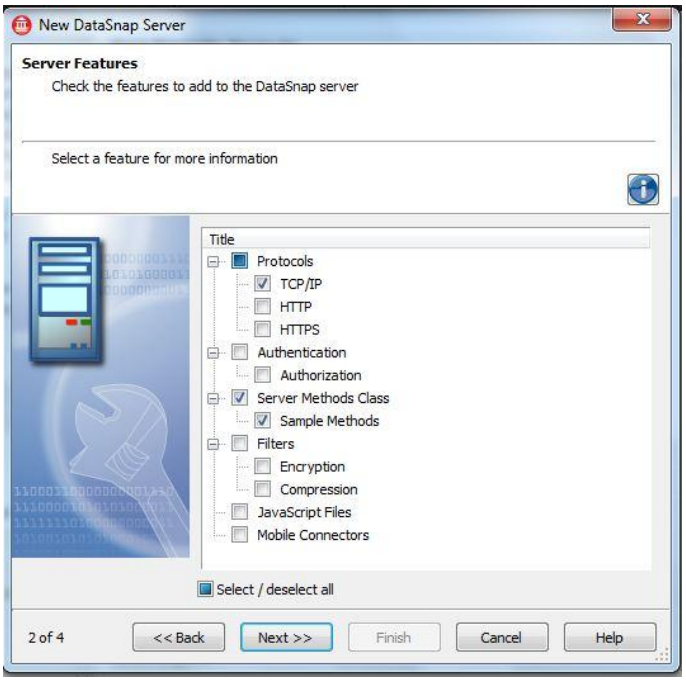
If you select the Sample Methods option, then the ServerMethodsUnit will contain the implementation of two simple methods called EchoString and ReverseString, which return the Value given as parameter in normal respective reversed states.

The Filters category specifies which filters the DataSnap Server will employ. You can chose to include an Encryption filter, a Compression filter, or both:

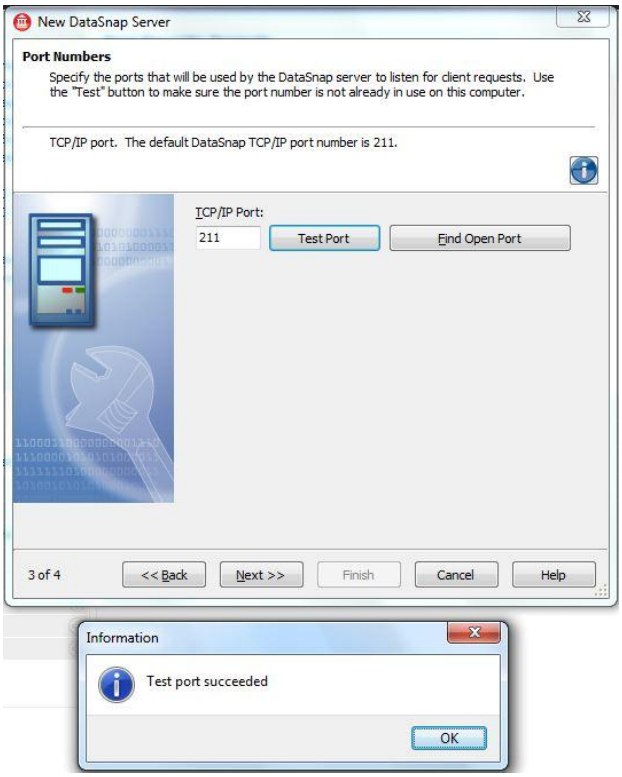
- The Encryption filter adds a PC1 and RSA filters for the selected communication protocols. Note that the RSA filter requires the OpenSSL libraries to be present on the server and on any client that will connect to this server.
- The Compression filter adds a ZLib compression filter for the selected communications protocols.
- The JavaScript files set up the project with the JavaScript framework and proxy generator.

If you select Mobile Connectors, your project supports proxy dispatching for applications on mobile devices such as Android, Windows 7 Phone, and iPhone. For more information, see DataSnap Connectors for Mobile Devices on the Embarcadero DocWiki at http://docwiki.embarcadero.com/RADStudio/en/Getting_Started_with_DataSnap_Mobile_Connectors.

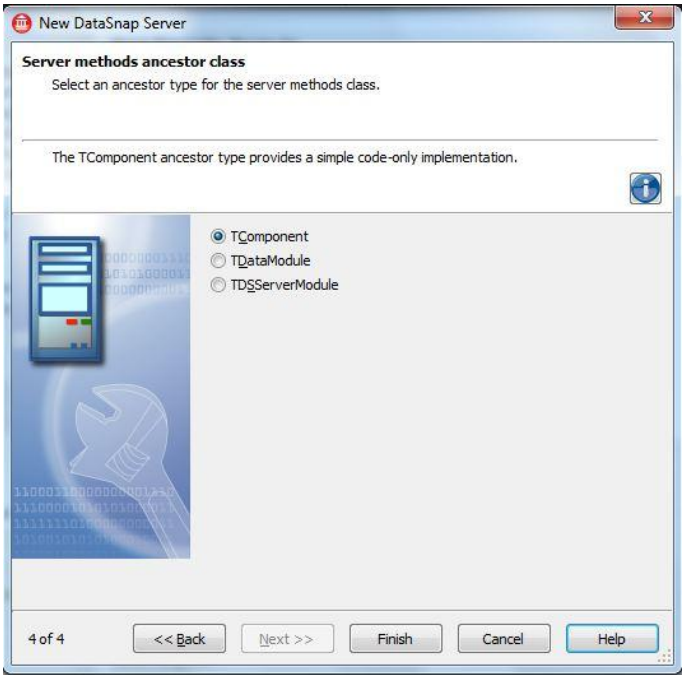
We'll keep it simple and just use TCP/IP and have the wizard generate sample methods for our client to use.



In the 3rd wizard step (step 3 of 4), the DataSnap wizard asks you for the Port number for your TCP/IP based DataSnap server. You can choose any port you want (or a port that is opened for you in your firewall). There is also "Test Port" button to make sure the port is actually available for use. There is a second button that you can use to find an open port. Note that if you select the HTTPS feature in the previous step, you have to specify the HTTPS communications port also. The same buttons as for the TCP/IP communication port are available: Test Port and Find Open Port. Leave the default choice for port 211 and click the "Test Port" button to make sure this port is available. If you get a "Test Port Succeeded" dialog box, then you are okay to proceed.

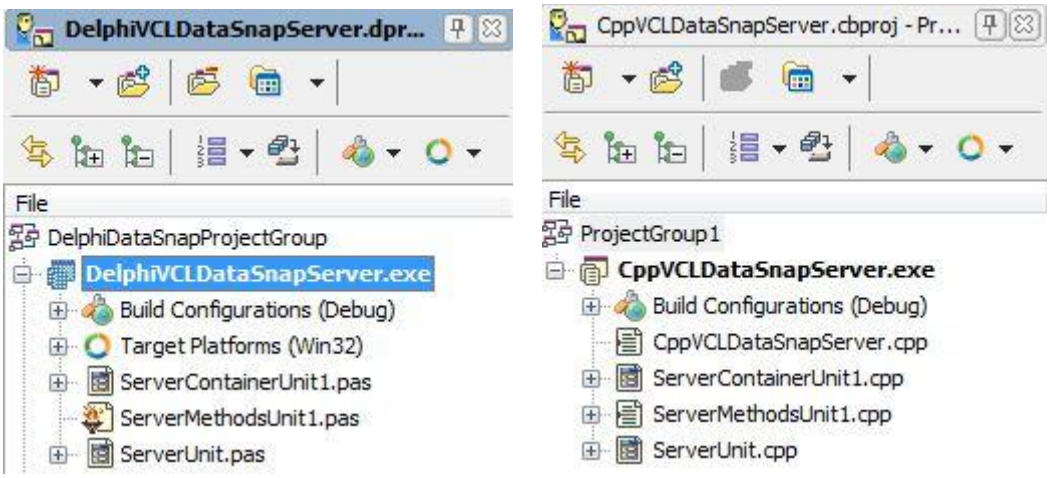


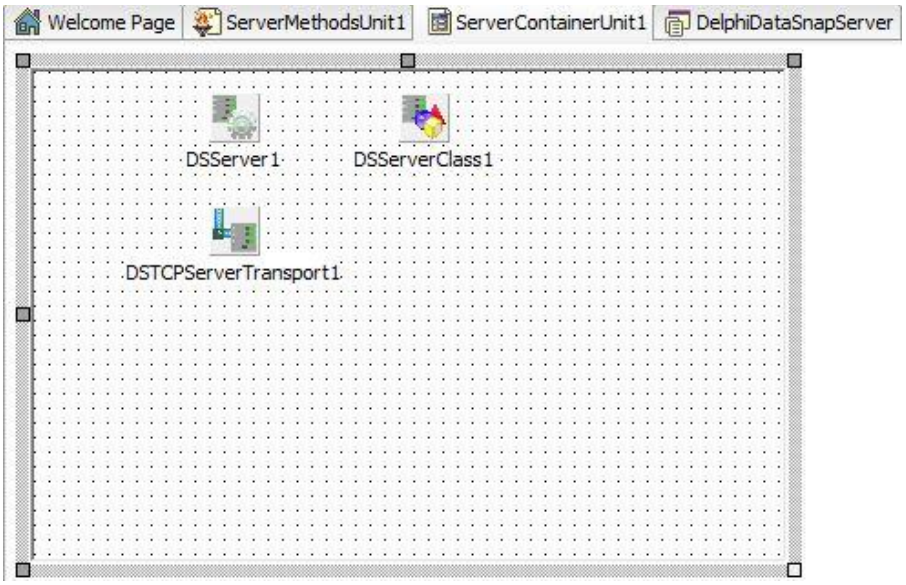
The final wizard step (step 4 of 4) lets you select an ancestor type for the server methods class. Choose TDSServerModule to expose datasets from the server to client applications. Choose TDataModule if you want to use non-visual components in your server class. Choose TComponent if you want to entirely implement the server class. For this first DataSnap server leave the default choice, TComponent.



If you chose to have additional features in the DataSnap wizard, you may see additional wizard steps.

Click the "Finish" button and you'll see that a DataSnap server application project is created with a ServerContainerUnit, ServerMethodsUnit and a Server Unit. Save the project to a folder and use "DelphiDataSnapServer" or "CppDataSnapServer" for the project name and use ServerUnit for the server unit name.





The ServerMethodsUnit contains the source code for the implementation of the sample DataSnap server methods generated by the wizard (if you chose that option):

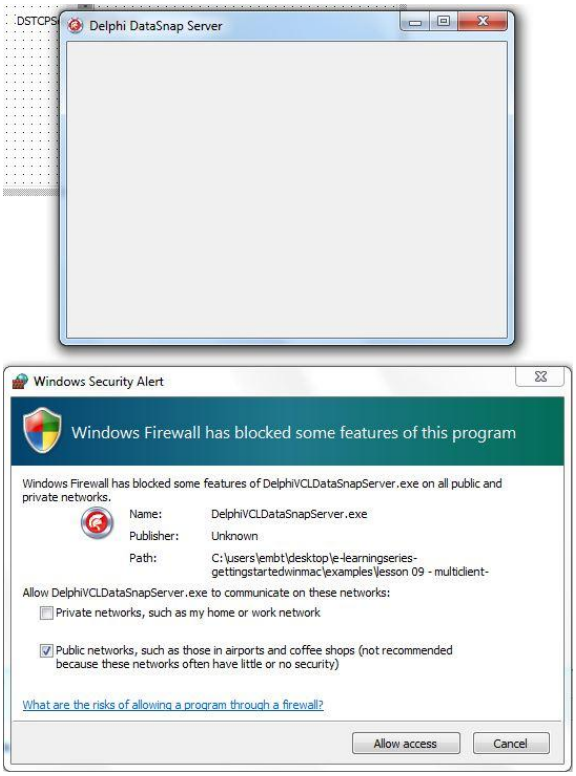
```
// Delphi - ServerMethodsUnit1.pas
unit ServerMethodsUnit1;

interface
uses System.SysUtils, System.Classes, Datasnap.DSServer,
     Datasnap.DSAuth;

type
{$METHODINFO ON}
```

```
  TServerMethods1 = class(TComponent)
  private
    { Private declarations }
  public
    { Public declarations }
    function EchoString(Value: string): string;
    function ReverseString(Value: string): string;
  end;
{$METHODINFO OFF}

implementation

uses System.StrUtils;

function TServerMethods1.EchoString(
    Value: string): string;
begin
  Result := Value;
end;
function TServerMethods1.ReverseString(
    Value: string): string;
begin
  Result := System.StrUtils.ReverseString(Value);
end;
end.

// C++ - ServerMethodsUnit1.cpp
//---------------------------------------------------------
#include <SysUtils.hpp>
#pragma hdrstop

#include "ServerMethodsUnit1.h"
//---------------------------------------------------------
#pragma package(smart_init)
//---------------------------------------------------------
System::UnicodeString TServerMethods1::EchoString(
    System::UnicodeString value)
{
    return value;
}
//---------------------------------------------------------
System::UnicodeString TServerMethods1::ReverseString(
    System::UnicodeString value)
{
    return ::ReverseString(value);
}
//---------------------------------------------------------

// C++ - ServerMethodsUnit1.h
//---------------------------------------------------------

#ifndef ServerMethodsUnit1H
#define ServerMethodsUnit1H
```

```
//---------------------------------------------------------
#include <Classes.hpp>
#include <DSServer.hpp>
//---------------------------------------------------------
class DECLSPEC_DRTTI TServerMethods1 : public TComponent
{
private:  // User declarations
public:   // User declarations
    System::UnicodeString EchoString(
        System::UnicodeString value);
    System::UnicodeString  ReverseString(
        System::UnicodeString value);
};
#endif
```

Your DataSnap server application is ready to compile and run on Windows. Set the Target Platforms node in the Project Manager view to Win32 (Delphi and C++) or Win64 (Delphi). Right-click on the DataSnap Server application and Choose **Run without Debugging** from the Context Menu to start the DataSnap Server application (this allows you to also run your DataSnap client application form the IDE).

When the DataSnap Server application starts you will see the server main window and you will also see (if you have Windows Firewall turned on) a Windows Security Alert. The Alert happens when the DataSnap server tries to use port 211.



Click the "Allow Access" button on the security alert window. Your DataSnap server application is now listening on port 211 for Client applications to call its methods. When you want to stop the DataSnap Server console application, hit ALT-F4 or click on the close application button.

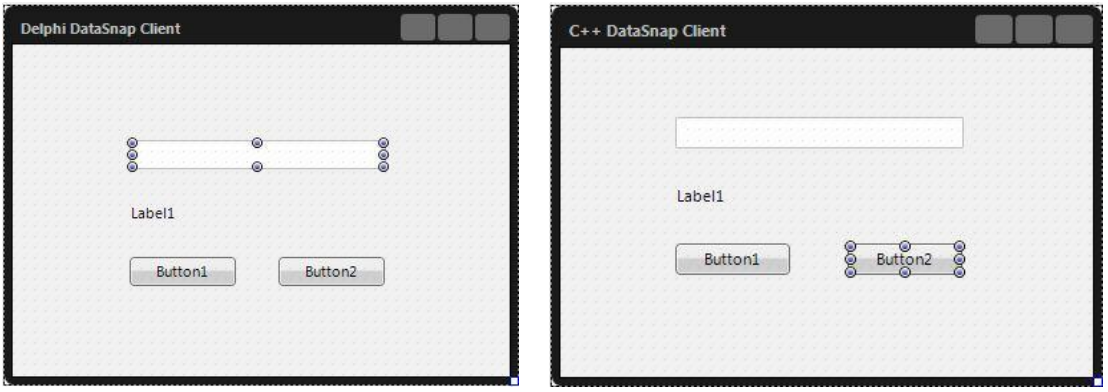
Step 2: Creating a Windows and Mac DataSnap Client Application

To create the client application in the same project group as the server application you just created, right-click on the name of your project group in the Project Manager and select **Add New Project** from the context menu. The New Items dialog box appears.

For Delphi, select the Delphi Projects category, then select **FireMonkey HD Application** and click OK. For C++, select the C++Builder Projects category, then select **FireMonkey HD Application** and click OK.

Set the form's **Caption** and **Name** properties to *Delphi DataSnap Client* and *DelphiDataSnapClientForm* or *C++ DataSnap Client* and *CppDataSnapClientForm*. Click the main menu item File > Save All. Save the unit file as DelphiClientUnit or CppClientUnit and save the project as DelphiDataSnapClient or CppDataSnapClient. Save the Project Group as DelphiDataSnapprojectGroup or CppDataSnapProjectGroup.

Next you'll create the UI for your Windows and Mac application. Add a TEdit, TLabel and two TButton(s) to your client application. Arrange and resize them so they look something like the following form:

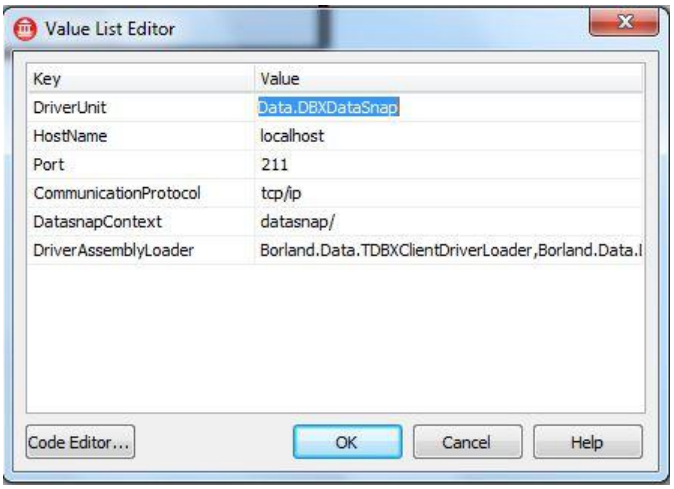


Set the **Text** and **Name** properties for Button1 to "Echo String" and "EchoStringButton". Set the **Text** and **Name** properties for Button2 to "Reverse String" and "ReverseStringButton".

Add a TSQLConnection component from the dbExpress category in the Tool Palette. Set the following properties for the TSQLConnection component using the Object Inspector:

- Driver: Datasnap (from the client's perspective, this provider looks like a connection to a database, but in fact provides connectivity to DataSnap servers.)
- LoginPrompt: False (optional, to prevent the user name and password dialog from appearing every time the client connects to the server.)
- If you chose a port number that is different from port 211 in the wizard, you'll need to use the Object Inspector, double click on the **Params** property and change the **Port** parameter. If you

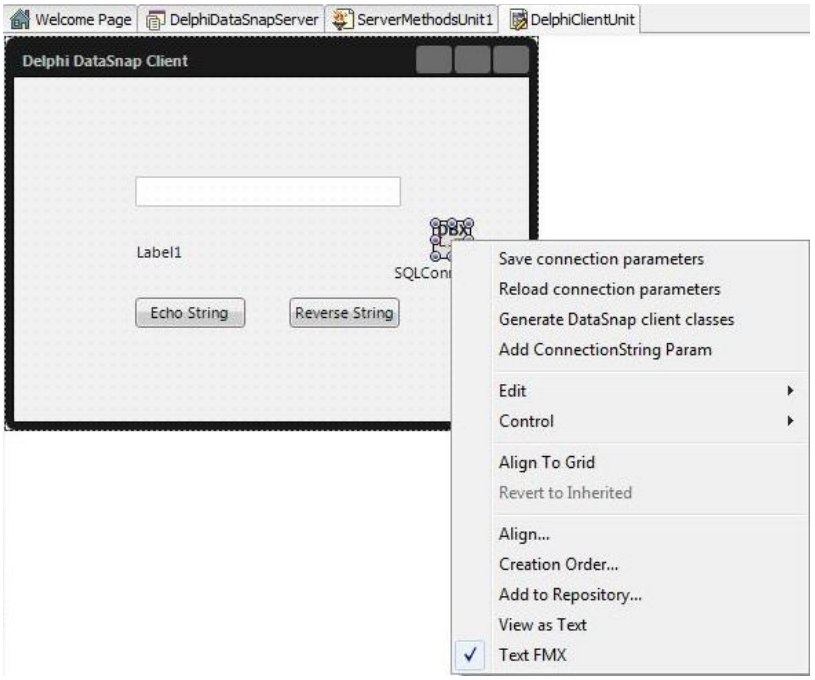
are running your DataSnap server on a different Windows machine you'll need to double click on the **Params** property and change the **HostName** parameter.



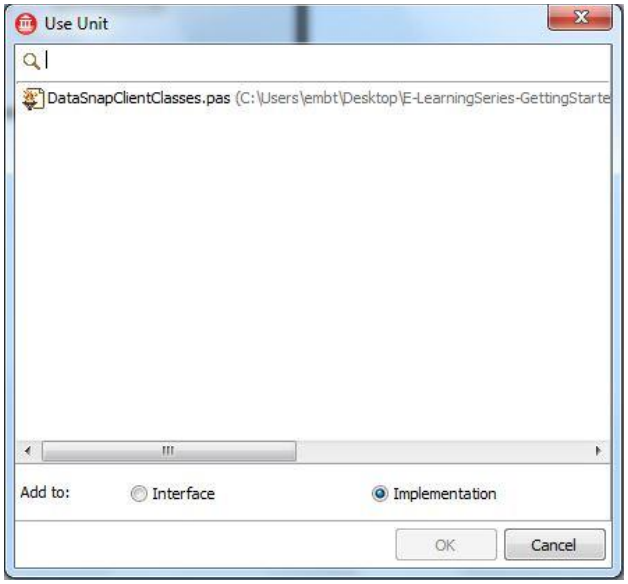
The DataSnap client application form should now look something like:

The most important step in creating the DataSnap Client application is creating the interface that contains the prototype of all functions implemented on the DataSnap server. Do this as follows:

- Activate the DataSnap server project by double-clicking the name of the server project in Project Manager.
- Choose **Run > Run Without Debugging** from the main menu to run the DataSnap server (unless it is already running).
- While the server is running, activate the DataSnap client project by double-clicking its name in the Project Manager.
- In the Design tab, set the **Connected** property of the TSQLConnection component to *True*. If this works, your DataSnap client is now connected to your DataSnap server at design time.
- Right-click the TSQLConnection component on the Client form and click "Generate DataSnap client classes" in the context menu. A new unit is added to your client project, containing information about classes implemented on the server and all the methods contained by these classes.



- For Delphi, save the new unit as DataSnapClientClasses.pas. Select the DelphiClientUnit form and then use **File > Use Unit,** select the unit and click the OK button to add the client classes unit's name (DataSnapClientClasses) to the implementation section of your DataSnapClient.pas file.



- For C++, save the new unit as DataSnapClientClasses.cpp. Add the following line to the beginning of the CppDataSnapClient.cpp file:

```
#include "DataSnapClientClasses.h"
```

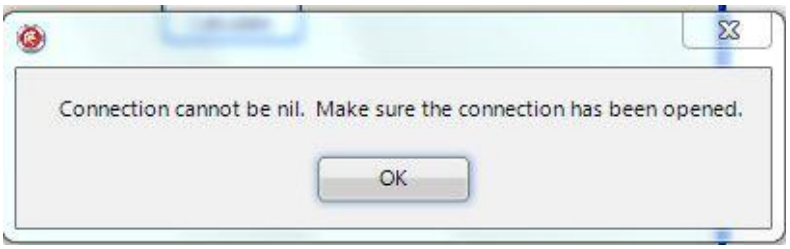
Add event handlers for each of the TButton(s) that will call the DataSnap server methods to echo and reverse the text typed into the TEdit box. Here is the code for each Button event handler (Delphi and C++ code):

```
// Delphi:
procedure TDelphiDataSnapClientForm.EchoStringButtonClick(
    Sender: TObject);
var
  Temp : TServerMethods1Client;
Begin
  Temp := TServerMethods1Client.Create(
    SQLConnection1.DBXConnection);
  try
    Label1.Text := Temp.EchoString(Edit1.Text)
  finally
    Temp.Free();
  end
end;

procedure TDelphiDataSnapClientForm.ReverseStringButtonClick(
    Sender: TObject);
Var
  Temp : TServerMethods1Client;
```

```
begin
  Temp := TServerMethods1Client.Create(
    SQLConnection1.DBXConnection);
  try
    Label1.Text := Temp.ReverseString(Edit1.Text)
  finally
    Temp.Free()
  end
end;


// C++:
//---------------------------------------------------------------------
void __fastcall TCppDataSnapClientForm::EchoStringButtonClick(
    TObject *Sender)
{
  TServerMethods1Client *Temp;
  Temp = new TServerMethods1Client(
    SQLConnection1->DBXConnection);
  try
  {
    Label1->Text = Temp->EchoString(Edit1->Text);
  }
  __finally
  {
    delete Temp;
  }
}
//---------------------------------------------------------------------
void __fastcall TCppDataSnapClientForm::ReverseStringButtonClick(
    TObject *Sender)

{
  TServerMethods1Client *Temp;
  Temp = new TServerMethods1Client(
    SQLConnection1->DBXConnection);
  try
  {
    Label1->Text = Temp->ReverseString(Edit1->Text);
  }
  __finally
  {
    delete Temp;
  }
}
//---------------------------------------------------------------------
```

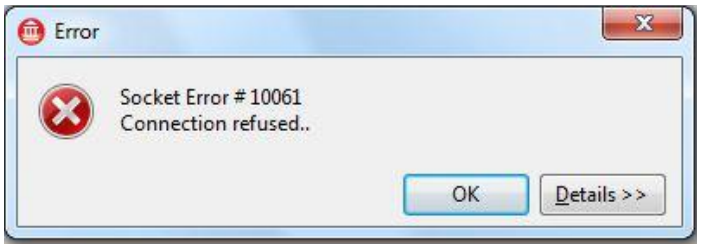
Finally, build and run the client application.

If you get an error message at runtime containing the text,



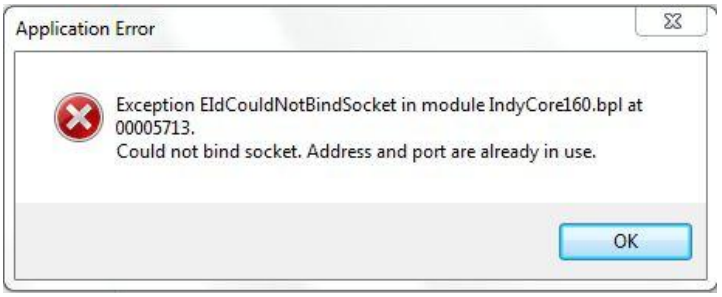
This can happen if your SQLConnection Connected property is not set to True. Make sure the SQLConnection's Connected property is set to True before running the DataSnap client application. A refinement to the client application would be to add a third button to turn on and off the SQLConnection's Connected property. Another way would be to, on FormCreate, make sure the Connected property is True and on FormDestroy to set the Connected property to False.

You might also get an error when you load your project group, with the DataSnap client project activated, with the SQLConnection Connected property set to True and without the DataSnap server application already running:



There is nothing to worry about if you get this error. The form designer is trying to talk with the DataSnap Server at design time and can't find it running on port 211. Click OK, start your DataSnap server application and re-activate the DataSnap client application.

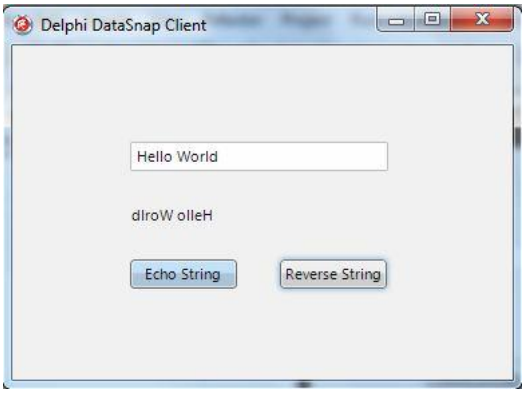
One final error you might see when you try to run a DataSnap server is "Address and port already in use."



This error can occur in at runtime or at project design time if two DataSnap servers are trying to use the same port number. To fix the problem shut down the DataSnap servers or if you need both running make sure they are both using different ports.

If the DataSnap Client application successfully connects to the DataSnap server, the client form is displayed. Enter some text in the edit box and click the Echo String and Reverse String buttons. The

event handlers will call the DataSnap server methods and set the Text property of the TLabel to the string that is returned.



To run the DataSnap client on Mac OS X you'll need to:

- Add a Target Platform node for OS X and make it the active platform.
- Set the SQLConnection's **Params** property **HostName** value to the computer name or IP address where your DataSnap server is running (the default HostName value is "localhost", on my Windows 7 guest OS under VMWare Fusion for the Mac the IP address 192.168.xx.xxx).
- If you want to control the Server and Client apps from the RAD Studio IDE, make sure your PAServer is running on the Mac.

Run the DataSnap client application. The IDE will deploy the DataSnap client application to the Mac using your remote profile settings. On the Mac, enter some text in the edit box and click the buttons to call the DataSnap server methods.

You can mix and match DataSnap clients and servers using Delphi and C++. You can mix and match Console applications, FireMonkey applications and VCL applications as well.

Before terminating the server application, make sure to close all the SQL connections. In this example, set the **Connected** property of the TSQLConnection component in the client to False. DataSnap does not

warn you about pending connections, so even if the server seems to close, it does not close until there are no connections to it. Closing all the client applications does not solve this problem either, because the Delphi IDE can automatically open a connection to the server and browse for the exposed classes and methods.

You will find another simple DataSnap server and client tutorial in the Embarcadero DocWiki at http://docwiki.embarcadero.com/RADStudio/en/Tutorial:_Using_a_DataSnap_Server_with_an_Application. This tutorial shows you how to create a DataSnap server method and DataSnap client application that sums two numbers. Pawel Glowacki, in his Delphi Labs for DataSnap episode 1, shows you how to create a simple DataSnap calculator server and client using Delphi. You'll find the article at http://edn.embarcadero.com/article/41176. You can download the project source code at http://cc.embarcadero.com/item/28184.

Creating a Windows and Mac Multi-Tier Database Application

Next, we will create a DataSnap based Multi-Client, Multi-Platform, Multi-Tier database application. The application will include:

- FireMonkey based DataSnap server that uses an InterBase database
- FireMonkey DataSnap client that provides the user interface for the database operations and can run on Windows and Mac.

The DataSnap client becomes a thin client which only needs the DataSnap connection to access the database. The DataSnap server has the database drivers and business logic to work with a local or remote database.

The DataSnap server application will only (currently) run on Windows using either Delphi or C++. You can build DataSnap client applications that run on Windows and Mac using Delphi and C++. With RAD Studio you can also build DataSnap client applications for RadPHP, JavaScript and using the DataSnap Mobile Connectors we support iOS (Objective-C and Delphi), Android (Java), Blackberry (Java), and Windows Phone (C#). The DataSnap client application does not need to be implemented in the same language as the DataSnap server.

Use the following steps to create the DataSnap server and client. Remember to do a **File > Save All** (shift-control-s) at the end of each step.

Step 1: Create a new FireMonkey DataSnap Server Project.

For Delphi, choose **File > New > FireMonkey HD Application – Delphi** to create a new Delphi project. For C++, choose **File > New > FireMonkey HD Application - C++Builder** to create a new C++Builder project.

Set the form's **Caption** property to "Delphi DB DataSnap Server Application" or "C++ DB DataSnap Server Application". Click **File > Save All** to save the project.

For Delphi, save the unit file as DelphiDBServerUnit.pas and save the project as DelphiDBDataSnapServer.dproj. For C++, save the file as CppDBServerUnit.cpp and save the project as CppDBDataSnapServer.cbproj.

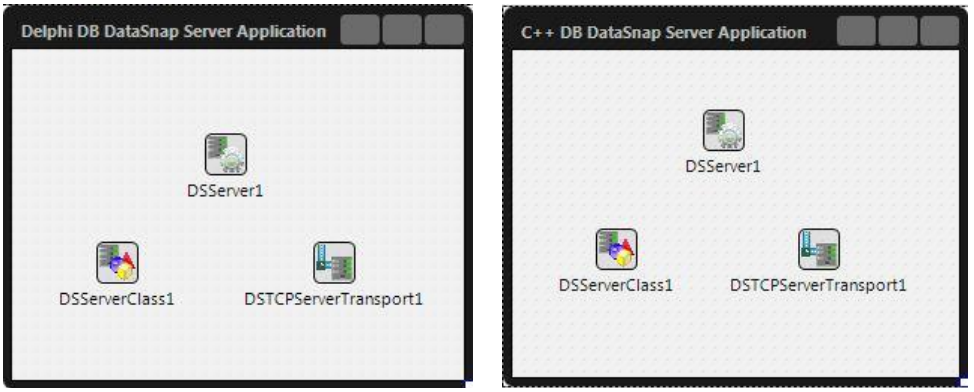
Step 2: Add the DataSnap Server components

Drag the following components from the Datasnap Server category of the Tool Palette onto the form:

- TDSServer
- TDSServerClass
- TDSTCPServerTransport

Click **File > Save All** to save the project.

Your server form should now look like this:



Step 3 – link the DataSnap server components together

Using the Object Inspector do the following to link the three DataSnap server components together:

Select the TDSServerClass component on the form. On the drop-down menu, set its Server property to the name of your TDSServer component, DSServer1 in this example.

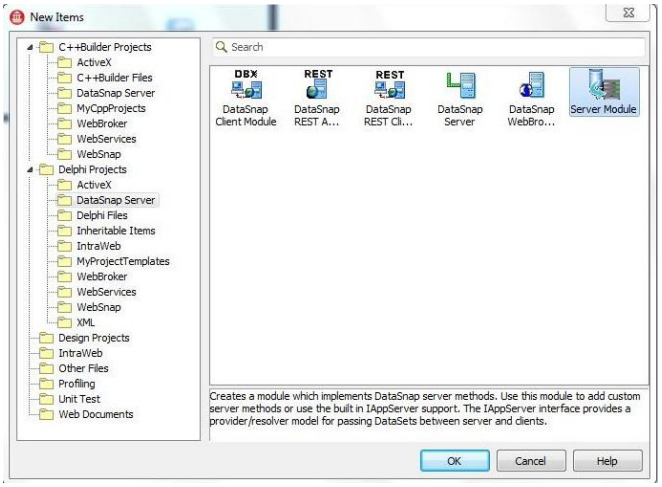
Select the TDSTCPServerTransport component on the form. On the drop-down menu, set its Server property to the name of your TDSServer component, DSServer1 in this example.

Click **File > Save All** to save the project.

Step 4 – Add a ServerModule to your project

Use **File > New > Other…** to bring up the "New Items" object gallery. You can also click on the "New Items" speed button on the tool bar (the page icon with a green + sign on it).

For a Delphi DataSnap Server, add a new Delphi file called a Server Module by clicking the tab **Delphi Projects > DataSnap Server**.



Select **Server Module** and click OK. After you click OK you will see the following warning dialog:

Since the Server Module can work with VCL, FireMonkey and common components, the IDE will display this warning as a reminder to limit your use to FireMonkey and common components. Just click the Yes button to continue.

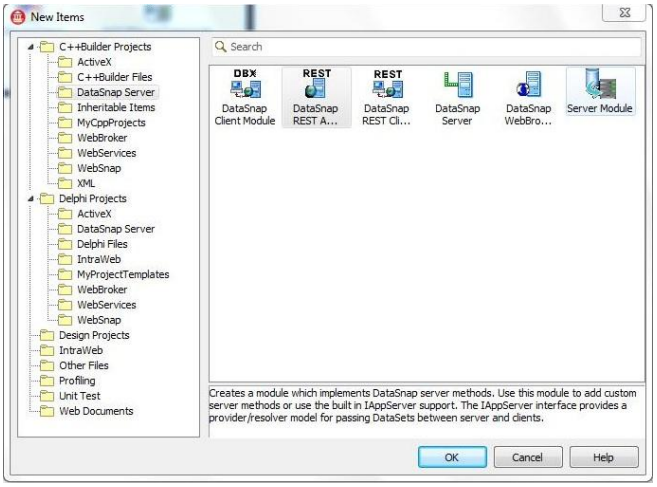
Using the Object Inspector set the **ClassGroup** property to FMX.Types.TControl.

The Server Module wizard will create a Server Module form and class Server Module class declaration. We'll be adding database access components in the form and additional public methods in the server class in the next step. The generated Delphi server module class declaration is:

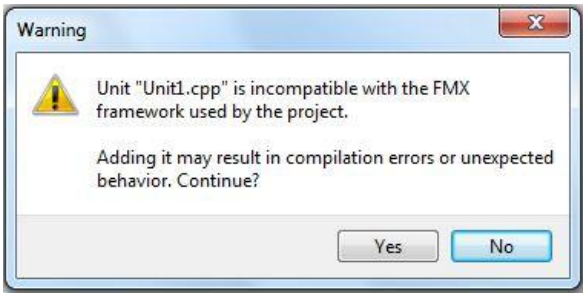
```
TDSServerModule1 = class(TDSServerModule)
private
  { Private declarations }
public
  { Public declarations }
end;
```

Save the module as DelphiServerModule.pas.

For a C++ DataSnap Server, add a new C++ file called a Server Module by clicking the tab C++Builder Projects > DataSnap Server.



Select **Server Module** and click OK. After you click OK you will see the following warning dialog:



Since the Server Module can work with VCL, FireMonkey and common components, the IDE will display this warning as a reminder to limit your use to FireMonkey and common components. Just click the Yes button to continue.

Using the Object Inspector set the **ClassGroup** property to FMX.Types.TControl.

The Server Module wizard will create a Server Module form and a Server Module class declaration. We'll be adding database access components in the form and additional public methods in the server class in the next step. The generated C++ server module class declaration is:

```
class TDSServerModule1 : public TDSServerModule
{
__published:    // IDE-managed Components
private:        // User declarations
public:         // User declarations
    __fastcall TDSServerModule1(TComponent* Owner);
};
```

Save the module as CppServerModuleUnit.cpp.

For both Delphi and C++ DataSnap servers, the TDSServerModule class inherits from TDSServerModuleBase which inherits from TProviderDataModule which inherits from TDataModule. You can add various database controls to the Server Module. You can also add public methods to your Server Module using the Code Editor.

Data modules are initially framework-neutral. In order to use framework-specific library elements, you need to set the platform affinity by selecting a framework-specific value for the ClassGroup pseudo-property:

- System.Classes.TPersistent, the default setting, indicates framework neutrality and includes only RTL elements that are not framework-specific.
- Vcl.Controls.TControl - sets the VCL framework, including RTL elements that are not framework-specific.
- FMX.Types.TControl - sets the FMX framework, including RTL elements that are not framework-specific.
- FMX_Types.TControl - sets the FMX iOS framework, including RTL elements that are not framework-specific.

Click **File > Save All** to save the project.

Step 5 – Add Database Components to your DataSnap Server Project

You can add components to a project by dragging items from the Data Explorer. In the **Project Manager view**, click the **Data Explorer** tab. If the INTERBASE item in the tree view isn't expanded, click the plus sign to expand it. Under INTERBASE, open the EMPLOYEE entry and then expand the Tables item. Drag the EMPLOYEE table to the DelphiServerModuleUnit form or CppServerModuleUnit form, which results in two new dbExpress components being added to the form:

- A TSQLConnection component. Using the Object Inspector, set its Name property to "EMPLOYEE_CONNECTION".
- A TSQLDataSet component. Using the Object Inspector, change its Name property to "EMPLOYEE_TABLE".

When you dragged the table to the form, the two components were automatically connected. When you set the Name property of the TSQLConnection component, you'll notice that the SQLConnection property of EMPLOYEE_TABLE was changed to the EMPLOYEE_CONNECTION.

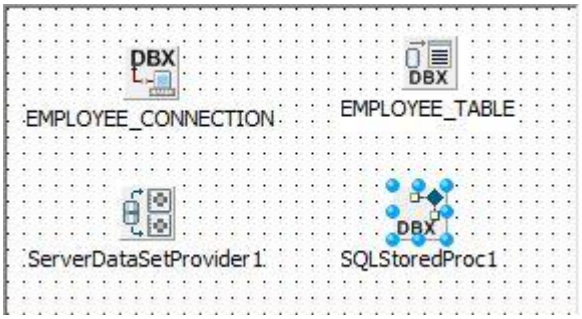
Place two additional components on the DelphiServerModuleUnit form or CppServerModuleUnit form:

- Add a TDataSetProvider. Using the Object Inspector, set its DataSet property to EMPLOYEE_TABLE on the drop-down menu. Change its name to ServerDataSetProvider1 to distinguish it from another TDataSetProvider that will be added later.

- Add a TSQLStoredProc component. Set its SQLConnection property to "EMPLOYEE_CONNECTION" using the property's drop-down menu. Set its StoredProcName property to "GET_EMP_PROJ" using the property's drop-down menu. "GET_EMP_PROJ" is one of the stored procedures in the Employee database. This stored procedure obtains a project ID associated with an employee number.

Click **File > Save All** to save the project.

The Server Module for Delphi or C++ projects should look like the figure below.



Step 6- Add Functions to Your Server Module

Add any functions to your Server Module that you want to expose as public. Any server methods in the public section of your Server Module may be called by a DataSnap client application that is connected to it.

For Delphi: with the DelphiServerModuleUnit selected, click the Code tab. In the type section for the ServerModule under the public section, add this function declaration:

```
function callStoredProcedure (mylocalkey : Integer) : String;
```

Use class completion by pressing CTRL-SHIFT-C to create a stub for this function in the implementation section.

For C++: with the CppServerModuleUnit selected, click the CppServerModuleUnit.h tab to display the header file. Add the following function under public:

```
String _fastcall callStoredProcedure (int mylocalkey);
```

After adding the database components and the public method, the Delphi and C++ Server Module classes are now declared as

```
// Delphi
TDSServerModule1 = class(TDSServerModule)
  EMPLOYEE_CONNECTION: TSQLConnection;
  EMPLOYEE_TABLE: TSQLDataSet;
  ServerDataSetProvider1: TDataSetProvider;
  SQLStoredProc1: TSQLStoredProc;
```

```
private
  { Private declarations }
public
  { Public declarations }
  function callStoredProcedure (mylocalkey : Integer) : String;
end;

// C++
class TDSServerModule1 : public TDSServerModule
{
__published:    // IDE-managed Components
  TSQLConnection *EMPLOYEE_CONNECTION;
  TSQLDataSet *EMPLOYEE_TABLE;
  TDataSetProvider *ServerDataSetProvider1;
  TSQLStoredProc *SQLStoredProc1;
private:        // User declarations
public:         // User declarations
  String _fastcall callStoredProcedure (int mylocalkey);
  __fastcall TDSServerModule1(TComponent* Owner);
};
```

Click **File > Save All** to save the project.

Step 7 - Write the Server Side Code for the Function You Just Added.

The callStoredProcedure function calls a stored procedure with an integer parameter employee number (EMP_NO). The function obtains an AnsiString project ID (PROJ_ID). The function sets the input parameter, executes the procedure, then retrieves the output parameter. The function parallels the function that will be written in the client in terms of parameters. We have already set the value of the StoredProcName property of the TSQLStoredProc component to the stored procedure name, "GET_EMP_PROJ".

For Delphi, add the following function to the DelphiServerModuleUnit.pas:

```
function TDSServerModule1.callStoredProcedure(
    mylocalkey: Integer): String;
var
  myString : String;
begin
  SQLStoredProc1.ParamByName('EMP_NO').AsInteger := mylocalkey;
  SQLStoredProc1.ExecProc;
  myString := SQLStoredProc1.ParamByName('PROJ_ID').AsString;
  result := myString;
end;
```

For C++, add the following function after the other member functions in CppServerModuleUnit.cpp:

```
String _fastcall TDSServerModule1::callStoredProcedure (
    int mylocalkey) {
  String myString;
  SQLStoredProc1->ParamByName("EMP_NO")->AsInteger = mylocalkey;
  SQLStoredProc1->ExecProc();
  myString = SQLStoredProc1->ParamByName("PROJ_ID")->AsString;
```

```
  return myString;
}
```

Note that since the actual stored procedure parameter names, EMP_NO and PROJ_ID, are used, their ordinal value is obtained by `ParamByName`.

Step 8 – Add the OnGetClass Event Handler for the DataSnap Server Class

Go back to DelphiDBServerUnit form or CppDBServerUnit form. Select the TDSServerClass component. In the Object Inspector for the TDSServerClass component, click the Events tab and double-click on the OnGetClass event. This event handler code determines which server class the DataSnap server uses:

```
// Delphi
procedure TForm1.DSServerClass1GetClass(
    DSServerClass: TDSServerClass;
    var PersistentClass: TPersistentClass);
begin
  PersistentClass := TDSServerModule1;
end;

// C++
void __fastcall TForm1::DSServerClass1GetClass(
    TDSServerClass *DSServerClass,
    TPersistentClass &PersistentClass) {
  PersistentClass = __classid(TDSServerModule1);
}
```

Note that the variable `PersistentClass` is assigned to a class reference—not an object reference.

Provide the linkage needed in ServerForm to the ServerModule.

For Delphi, go to the `uses` section of the ServerForm unit and add DelphiServerModuleUnit, so that TDSServerModule1 is recognized. (You can also use **File > Use Unit**).

For C++, add this line after the other includes in CppDBServerUnit.cpp:

```
#include "CppServerModuleUnit.h"
```

Save the unit. Build the server project (**Project > Build** or Shift-F9) and fix any errors, but do not run the server at this time.

Click on the Project Manager view in RAD Studio. Save the project group by right-clicking the project group and clicking Save Project Group. Save the project group as DataSnapDBApplication.groupproj. In the next section we will add another project to this project group.

This completes the server, which does two things:

- Provide database data that can be updated
- Execute a stored procedure and return a value

Next, you will create a DataSnap client that uses the server. The client exercises both functions of the server:

- Call a stored procedure and return a value.
- Display and update data in a database table.

Click **File > Save All** to save the project and project group.

Step 9 – Create the Client Application Project

To create the client application in the same project group as the server application, right-click the name of the project group in the Project Manager and select Add New Project or choose Project > Add New Project. The New Items dialog box will appears.

For Delphi, select the Delphi Projects category, then select **FireMonkey HD Application**. For C++Builder, select the C++Builder Projects category, then select **FireMonkey HD Application**.

Click OK. Select the new form and set its Caption property to "Delphi DB DataSnap Client Application" or "C++ DB DataSnap Client Application" in the Object Inspector.

Choose **File > Save All** to save the files:

For Delphi, save the unit as DelphiDBClientUnit.pas. Save the project as DelphiDBDataSnapClient.dproj. For C++, save the unit as CppDBClientUnit.cpp. Save the project as CppDBDataSnapClient.cbproj.

Double-click the CppDBDataSnapServer.cbproj.exe in the Project Manager view (to activate that project node). Choose **Run > Run Without Debugging** (or hit Shift-Control-F9) to run the DataSnap server application. You can minimize the server application Form that appears.

Note: You need to have the server running to be able to connect to the server and generate the DataSnap client classes in the next step.

Click **File > Save All** to save the project and project group.

Step 10 – Connect the DataSnap Client to the DataSnap Server

Place a TSQLConnection component on the new form and set its properties using the Object Inspector:

- Set the Driver property to "DataSnap". In Object Inspector, clicking on the **+** sign on the left of the Driver property to display (and allow you to set) Driver sub-properties:
    - Set Port to "211" (default).
    - Set HostName to "localhost" (default). Use localhost for testing the DataSnap server and client on the same machine. When you put the DataSnap server on a different

machine, then you will need to change the client's TSQLConnection HostName to the computer name or TCP/IP address where the DataSnap server is running.
    - These properties (and others) can also be set by changing the Params property. Click on the ellipsis (...) button in the Params property to display the Value List Editor. You can enter property values in this dialog and then click OK to set the values.
- Set the LoginPrompt property to false to prevent the user name and password dialog appearing every time the client connects to the server
- Set the Connected property to true

While the DataSnap server is running, right-mouse click on the TSQLConnection and select "**Generate DataSnap client classes**" from the context menu. This action creates a new unit with code that supports connecting and using the functions of your DataSnap server. For Delphi, save the generated unit as DelphiDBClientClasses.pas. For C++, save the generated unit as CppDBClientClasses.cpp.

Link the Delphi or C++ client classes unit to DataSnap DB client unit.

For Delphi, click on the DelphiDBClientUnit tab, click its Code tab, and then add DelphiDBClientClasses to the DelphiDBClientUnit's uses clause (or use **File > Use Unit**).

For C++, click on the CppDBClientUnit.cpp tab at the top of the Code Editor, then click on the Code tab at the bottom of the Code Editor. Add the following include after the other includes in CppDBClientUnit.cpp:

```
#include "CppDBClientClasses.h"
```

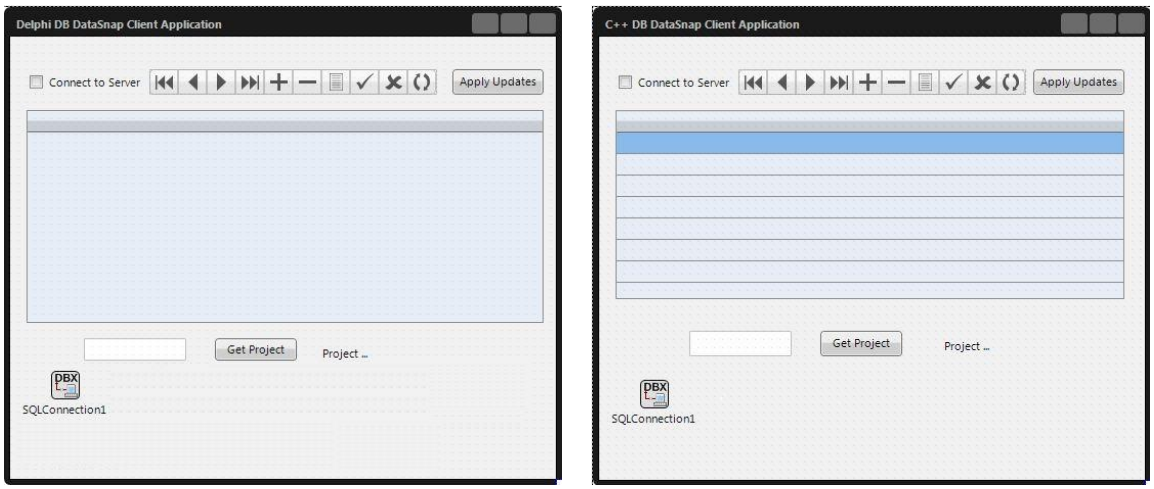
Click **File > Save All** to save the project and project group.

Step 11 – Start Building the DataSnap Client Application User Interface

Next we'll start building the DataSnap Client Application's user interface. In the DelphiDBClientUnit form, click the Design tab and drag components from the Tool Palette onto the form:

- A TBindNavigator control to navigate through the database.
- A TStringGrid control to view a database table.
- A TCheckBox control to open/close the DataSnap server connection and activate/deactivate the ClientDataSet.
- Two TButtons to update any changed database data and call the stored procedure. Set the corresponding TButton's Text properties to "Apply updates" and "Get project". Also change each associated TButton's Name property to "ApplyUpdatesButton" and "GetProjectButton".
- A TEdit control for the employee number.
- A TLabel control to show the project ID.

After you have placed the components, move and resize them as needed. Your DataSnap client form should look something like the following:

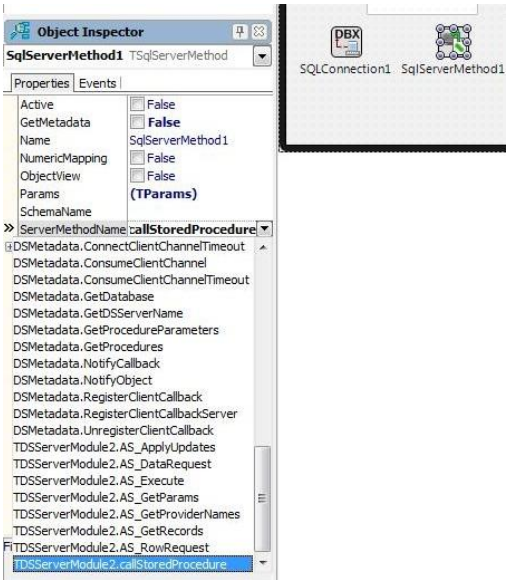


Step 12- Add Client Side Database Components

In this step you will add the database components and code that will connect your DataSnap client application to the database operations in your DataSnap server. Make sure that your DataSnap server is running.

Use IDE Insight or the Tool Palette, place a TSQLServerMethod component on the form. Set its SQLConnection property to "SQLConnection1" using the Object Inspector.

Set TSQLServerMethod's ServerMethodName property using the Object Inspector. When the DataSnap server is running, you can use the property's drop-down menu to see all the server methods available in your DataSnap server. The list will include DataSnap administration, metadata and server module methods. Select the "TDSServerModule1.callStoredProcedure", which is the function you previously created in the DataSnap server that calls the InterBase Employee database's "Get_EMP_PROJ" stored procedure.



Do not set TSQLServerMethods's Active property to true. If you do, you get an error message, because the "Get_EMP_PROJ" stored procedure does not return a dataset, it just returns the project name string. We will use code to call the method and put the returned string in the TLabel's Text property.

Add a TDSProviderConnection component onto the form. This provider component gives us the ability to freely navigate and resolve database updates with the DataSnap server. Set the SQLConnection property to "SQLConnection1". Set the ServerClassName property to the name of your DataSnap server's Server Module class name, which in our example is "TDSServerModule1".

Add a TClientDataSet onto the form. Set the ProviderName property to "DataSetProvider1" from the drop-down menu. Set the RemoteServer property to "DSProviderConnection1" from the drop-down menu. Set the ProviderName property to "ServerDataSetProvider1" from the drop-down menu.

Add a "TDataSource" component to the form and set its "DataSet" property to "ClientDataSet1".

Right mouse-click on the TStringGrid on the form and choose "Link to DB DataSource" from the context pop-up menu. Select DataSource1 from the "select a data source" list and click the OK button.

Two additional components will appear on your client form:

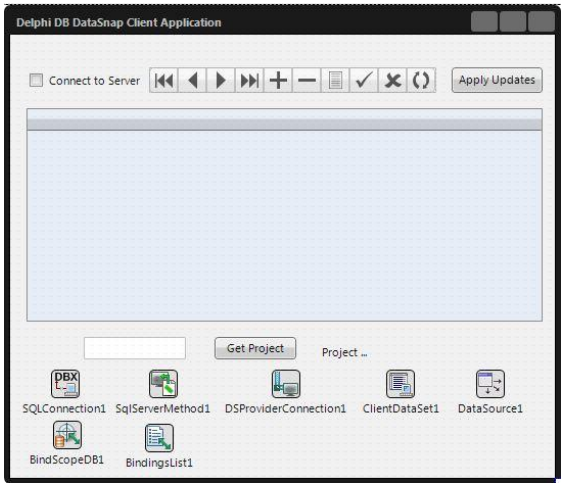
- BindScopeDB - Non-visual component that provides a way to make data contained by the specified data source available to all components that want to access it, using LiveBindings.
- BindingsList - Non-visual component that contains all of the LiveBindings expressions, methods and output converters.

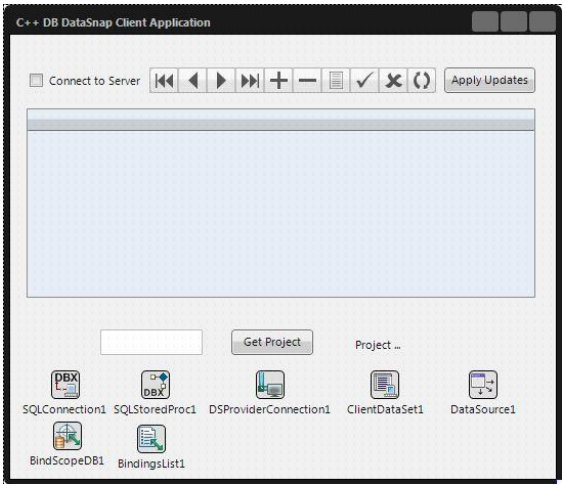
Select the BindDBNavigator component on the form. Set its BindScope property to "BindScopeDB1" from the drop down list in the Object Inspector.

Set the ClientDataSet's Active property to True. You will see live employee data in the TStringGrid. The data and metadata is coming via the DataSnap connection to your DataSnap server which is getting the data and metadata from the InterBase database.

Set ClientDataSet's Active property back to False. The client application will toggle the ClientDataSet's Active property using the CheckBox's OnChange event handler that we add in the next step.

After you've added the client side DataSnap access and ClientDataSet components and set the properties your DataSnap client, your form should look something like the following:





Step 13 – Add Event Handlers for the CheckBox and Apply Updates button

First add the OnChange event handler for the Connect to Server Checkbox. Select Checkbox and in the Events tab of the Object Inspector, double-click the OnChange event to generate the skeleton code. Add the following code which will open/close the DataSnap server connection and activate/deactivate the ClientDataSet:

```
// Delphi
procedure TForm1.CheckBox1Change(Sender: TObject);
begin
  // if Connect to Server is checked then
  //   open the DataSnap server connection
  //   activate the ClientDataSet
  if CheckBox1.IsChecked then begin
    SQLConnection1.Connected := True;
    ClientDataSet1.Active := True
  end
  // if Connect to Server is not checked then
  //   deactivate the ClientDataSet
  //   close the DataSnap server connection
  else begin
    ClientDataSet1.Active := False;
    SQLConnection1.Connected := False
  end;
end;
```

```
// C++
void __fastcall TForm1::CheckBox1Change(TObject *Sender)
{
  // if Connect to Server is checked then
  //   open the DataSnap server connection
  //   activate the ClientDataSet
  if (CheckBox1->IsChecked) {
    SQLConnection1->Connected = True;
    ClientDataSet1->Active = True;
  }
  // if Connect to Server is not checked then
  //   deactivate the ClientDataSet
  //   close the DataSnap server connection
  else {
    ClientDataSet1->Active = False;
    SQLConnection1->Connected = False;
  }
}
```

Add the `OnClick` event handler for the "Apply updates" TButton. Create the skeleton for the event handler as above and add this code:

```
// Delphi
procedure TForm1.ApplyUpdatesButtonClick(Sender: TObject);
begin
  ClientDataSet1.ApplyUpdates(0);
end;

// C++
void __fastcall TForm1:: ApplyUpdatesButtonClick(TObject *Sender)
{
  ClientDataSet1->ApplyUpdates(0);
}
```

As you are making changes to the data in the StringGrid, the changes are being stored in the ClientDataSet's Delta public (not published) property. Delta contains only information about those records inserted, modified, or deleted through the client. When the client dataset is linked to a provider, Delta is passed as an argument to the ApplyUpdates and Reconcile methods, which use the information in the change log to update the database. On return from successful application of updates, Delta is cleared.

ApplyUpdates takes a single parameter, MaxErrors, which indicates the maximum number of errors that the provider should tolerate before aborting the update process. If MaxErrors is 0, then as soon as an update error occurs, the entire update process is terminated. No changes are written to the database, and the client dataset's change log remains intact. If MaxErrors is -1, any number of errors is tolerated, and the change log contains all records that could not be successfully applied. If MaxErrors is a positive value, and more errors occur than are permitted by MaxErrors, all updates are aborted. If fewer errors occur than specified by MaxErrors, all records successfully applied are automatically cleared from the client dataset's change log.

ApplyUpdates returns the number of actual errors encountered, which is always less than or equal to MaxErrors plus one. This return value indicates the number of records that could not be written to the database.

The client dataset's ApplyUpdates method does the following:

- It indirectly calls the provider's ApplyUpdates method (in this case via TCP/IP from the DataSnap client to the DataSnap server). The provider's ApplyUpdates method writes the updates to the database and attempts to correct any errors it encounters. Records that it cannot apply because of error conditions are sent back to the client dataset.
- The client dataset 's ApplyUpdates method then attempts to reconcile these problem records by calling the Reconcile method. Reconcile is an error-handling routine that calls the OnReconcileError event handler. You must code the OnReconcileError event handler to correct errors.
- Finally, Reconcile removes successfully applied changes from the change log and updates Data to reflect the newly updated records. When Reconcile completes, ApplyUpdates reports the number of errors that occurred.

There are two events that let you handle errors that occur during the update process on the DataSnap server side and DataSnap client side:

- On the DataSnap server side: During the update process, the dataset provider generates an OnUpdateError event every time it encounters an update that it can't handle. If you correct the problem in an OnUpdateError event handler, then the error does not count toward the maximum number of errors passed to the ApplyUpdates method.
- On the DataSnap client side: After the entire update operation is finished, the client dataset generates an OnReconcileError event for every record that the provider could not apply to the database server.

In this “getting started” tutorial, I do not deal with errors and the ApplyUpdates parameter is set to 0 in the above event handler. In your real world applications you should always code an OnReconcileError or OnUpdateError event handler, even if only to discard the records returned that could not be applied. The event handlers for these two events work the same way. They include the following parameters:

- DataSet: A client dataset that contains the updated record which couldn't be applied. You can use this dataset's methods to get information about the problem record and to edit the record in order to correct any problems. In particular, you will want to use the CurValue, OldValue, and NewValue properties of the fields in the current record to determine the cause of the update problem. However, you must not call any client dataset methods that change the current record in your event handler.
- E: An object that represents the problem that occurred. You can use this exception to extract an error message or to determine the cause of the update error.
- UpdateKind: The type of update that generated the error. UpdateKind can be ukModify (the problem occurred updating an existing record that was modified), ukInsert (the problem occurred inserting a new record), or ukDelete (the problem occurred deleting an existing record).

- Action: A parameter that indicates what action to take when the event handler exits. In your event handler, you set this parameter to
    - Skip this record, leaving it in the change log. (rrSkip or raSkip)
    - Stop the entire reconcile operation. (rrAbort or raAbort)
    - Merge the modification that failed into the corresponding record from the server. (rrMerge or raMerge) This only works if the server record does not include any changes to fields modified in the client dataset's record.
    - Replace the current update in the change log with the value of the record in the event handler, which has presumably been corrected. (rrApply or raCorrect)
    - Ignore the error completely. (rrIgnore) This possibility only exists in the OnUpdateError event handler, and is intended for the case where the event handler applies the update back to the database server. The updated record is removed from the change log and merged into Data, as if the provider had applied the update.
    - Back out the changes for this record on the client dataset, reverting to the originally provided values. (raCancel) This possibility only exists in the OnReconcileError event handler.
    - Update the current record value to match the record on the server. (raRefresh) This possibility only exists in the OnReconcileError event handler.

For more information about ClientDataSet, you should check out Cary Jensen's excellent book, "Delphi in Depth: ClientDataSets" - http://www.jensendatasystems.com/cdsbook/. You can also watch (or download) Cary Jensen's series of blog posts about ClientDataSet at http://caryjensen.blogspot.com/search?q=clientdataset

Step 14 – Add Button Event Handler Code for the Stored Procedure Call

This step will show you how to create TButton event handler that will use the TEdit box and GetProject TButton to take an employee's number and make a DataSnap server method call that will execute the InterBase database's GET_EMP_PROJ stored procedure and return the Project ID and display it in the TLabel's Text property.

Create an event handler skeleton for the "Get Project" TButton's OnClick event. Clicking this TButton results in calling the method we defined on the server. Since the stored procedure takes an integer value, we need to convert the text in the employee number TEdit to an integer, call the server method which will in turn call use the database stored procedure to return the Project ID string. Here is the event handler code:

```
// Delphi
procedure TForm1.GetProjectButtonClick(Sender: TObject);
var
  mykey : Integer; //variable to hold text from edit box
  myServer : TDSServerModule1Client;  //server proxy we will call
  projectIDString : string;   // returned from stored proc
begin
  // If DataSnap server is connected call the remote method
  if SQLConnection1.Connected then begin
    mykey := StrToInt(Edit1.Text);  //conversion to integer
    // Server creation using the SQLConnection for communication
```

```
    myServer := TDSServerModule1Client.Create(
      SQLConnection1.DBXConnection);
    try
      // Calling method that calls stored procedure with the key
      // Save value returned from stored procedure
      projectIDString := myServer.callStoredProcedure(mykey);
      // if returned string is empty then display no projectID
      if projectIDString = '' then
        Label1.Text := '* NoProjID *'
      else
        Label1.Text := projectIDString
    finally
      myServer.Free  //free up the server
    end
  end
end;

// C++
void __fastcall TForm1::GetProjectButtonClick(TObject *Sender)
{
  // If DataSnap server is connected call the remote method
  if (SQLConnection1->Connected) {
    int mykey; //variable to hold text from edit box
    String projectIDString;   // returned from stored proc
    TDSServerModule1Client *myServer;  //server proxy we'll call
    mykey = StrToInt(Edit1->Text);  //conversion to integer
    // Server creation using the SQLConnection
    myServer = new TDSServerModule1Client(
      SQLConnection1->DBXConnection);
    try {
      // Calling method that calls the stored proc with the key.
      // Save value returned from stored procedure.
      projectIDString = myServer->callStoredProcedure(mykey);
      // if returned string is empty then display "* noProjID *"
      if (projectIDString == "")
        Label1->Text = "* NoProjID *";
      else
        Label1->Text = projectIDString;
    }
    __finally {
      delete myServer;  //free up the server
    }
  }
}
```

Notice that the preceding code sets the Text property of the TLabel to the value returned from calling the stored procedure: the project ID. If the stored procedure doesn't find a project ID for the employee (null string), then display "* NoProjID *" text in the TLabel's Text property.

Select **File > Save All** to save all the modified source files, projects and the project group.

Step 15 – Build and Run the Windows DataSnap Client Application

You are now ready to build and run the Windows client side of the project. Build the client project by right-clicking DelphiDBDataSnapClient or CppDBDataSnapClient in the Project Manager view and select Build. Fix any errors you find.

Run the client application. Click the "Connect to Server" CheckBox, which

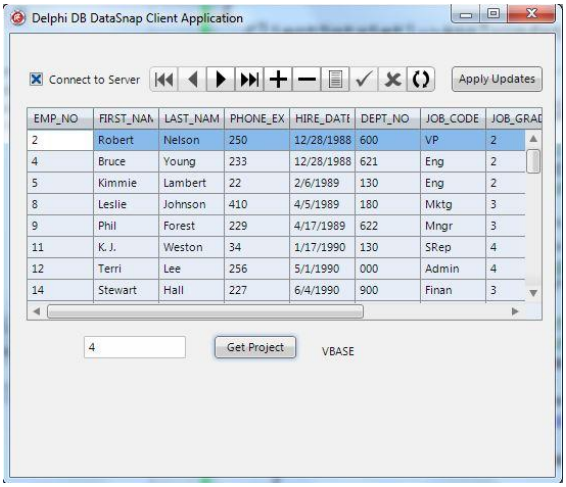
- If checked, connects to the DataSnap server and activates the ClientDataSet
- If unchecked deactivates the ClientDataSet and disconnects from the DataSnap server.

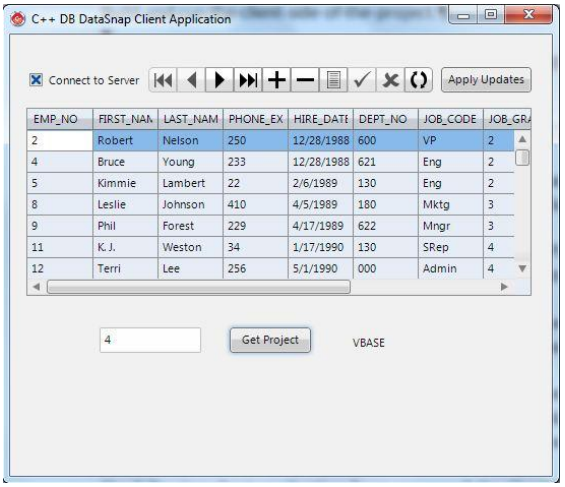
The TStringGrid gets populated with entries from the EMPLOYEE table. The TBindNavigator control is also active, allowing you to navigate through the table entries.

You can select a cell in the TStringGrid and change its value. Click the "Apply Updates" TButton and database table will be updated with any changes you make.

Finally, test the stored procedure. Enter one of the valid employee numbers in the TEdit control and click the "Get project" TButton. The text of the label should change to the appropriate project ID from the EMPLOYEE_PROJECT table, if one exists, otherwise it will display "* No ProjectID *":

The following client application form should appear as follows if the CheckBox is checked and you've tested the stored procedure:



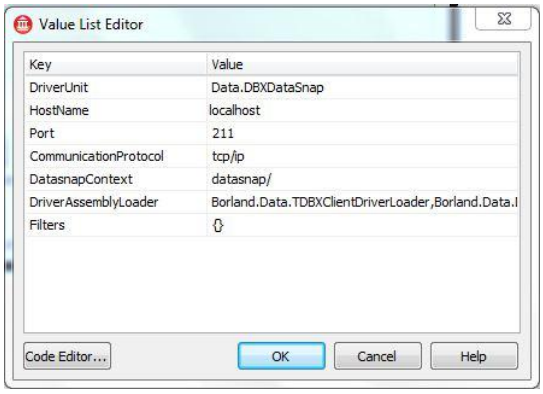


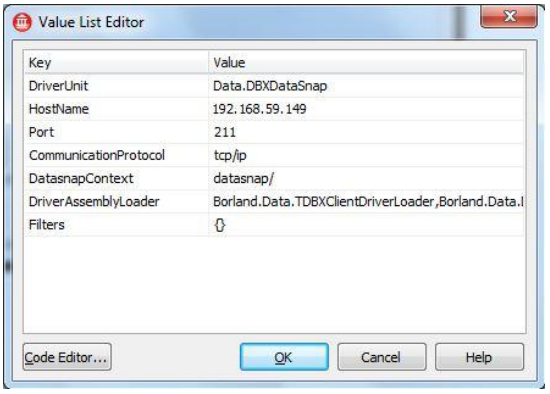
Step 16 – Build and Run the Mac DataSnap Client Application

As I mentioned earlier, you can create a Mac version of your DataSnap client application.

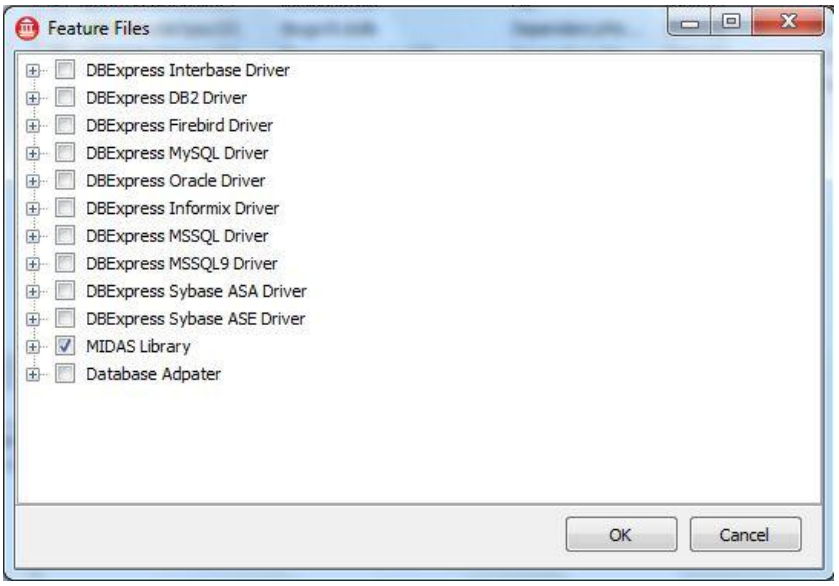
In the Project Manager view, right mouse-click on the Target Platforms node for your DataSnap client and add "OS X" as a target. Also make sure you have set up a remote profile for the target platform. Also make sure the PAServer is running on your Mac (the IDE needs this running to be able to deploy the built application to the Mac). You can also use these same steps to build and deploy your DataSnap client to remote Windows targets.

You also need to change the "HostName" parameter for the SQLConnection component in the client from localhost to the computer network name or TCP/IP address where you DataSnap server is running. Use the Object Inspector for the SQLConnection1 component and double click on the Params property to bring up the Value List Editor. You can skip this step if you changed or created a new DataSnap connection with the DataSnap dbExpress connection settings that already point to your DataSnap server. In this case, we used the default DataSnap connection name so the default HostName is "localhost". Change the HostName value to Computer name or TCP/IP address for your setup.

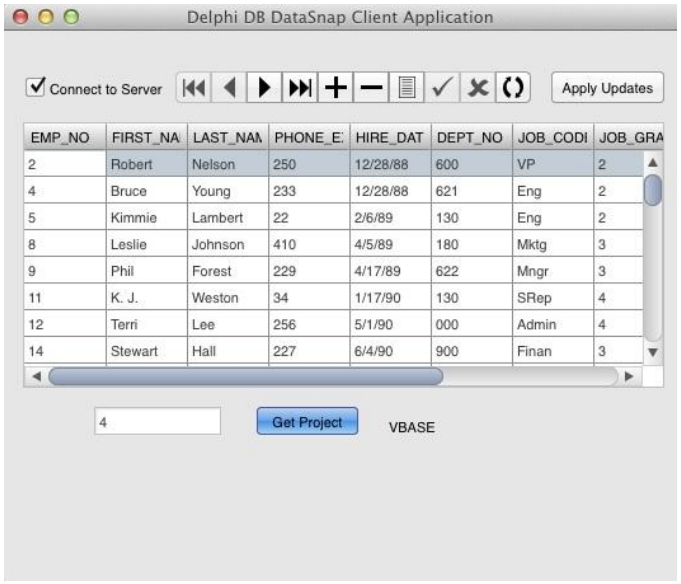




One final setting before you can build and deploy your DataSnap client application to the Mac, add the Midas Library Feature File to the deployment options for your client application project as the client uses ClientDataSet. Use **Project > Deployment** and click on the Feature Files icon and select MIDAS library (Project Deployment is also covered in Lesson 3 – the IDE). This will tell the IDE to deploy the MIDAS.dylib along with your Mac DataSnap client executable so that you can test the Mac version.



Choose Run without Debugging (Shift-Control-F9) or Run (F9) to compile and deploy the DataSnap client application to a Mac. Test the client on your Mac to make sure it can work with the DataSnap server running on Windows. The Mac DataSnap client application should work just like the Windows client worked.



[C++ app bitmap here]

Congratulations! You've build your first Multi-Client, Multi-Platform, Multi-Tier DataSnap Database application for Windows and Mac! ☺

There is always more to do and more to learn, especially with DataSnap multi-tier application development. For example, I did not check to see if there were any updates that should be applied if you've made changes and you un-check the CheckBox. You can always call ApplyUpdates before deactivating the ClientDataSet and closing the DataSnap server connection. Calling ApplyUpdates, when there are no updates to apply, will not try to send anything to the server, nor will it cause a runtime error.

You can also build both Delphi and C++ DataSnap server applications and/or DataSnap client applications and since they have the same functionality, you can mix and match one server with a different client.

As you learn more, you can also build DataSnap client applications using RadPHP - and mobile client applications using the DataSnap mobile connector technology - http://docwiki.embarcadero.com/RADStudio/en/DataSnap_Connectors_for_Mobile_Devices.

There are additional videos and articles about building DataSnap applications including:

- Delphi Labs: DataSnap XE - Multitier Database Application - http://edn.embarcadero.com/article/41189
- DataSnap XE2 – New Features and Improvements - http://cc.embarcadero.com/Item/28542
- REST and Mobile DataSnap Client Development - http://cc.embarcadero.com/Item/28543
- Creating Pure REST Servers using DataSnap - http://cc.embarcadero.com/Item/28561

Creating REST, WebBroker and Service based DataSnap Server Applications

Instead of building a FireMonkey HD, VCL Form, Console or Service based DataSnap application server you can also create a DataSnap REST Server Application and a DataSnap WebBroker Server Application. These types of DataSnap server applications are also easy to build but go beyond the scope of this "Getting Started" course.

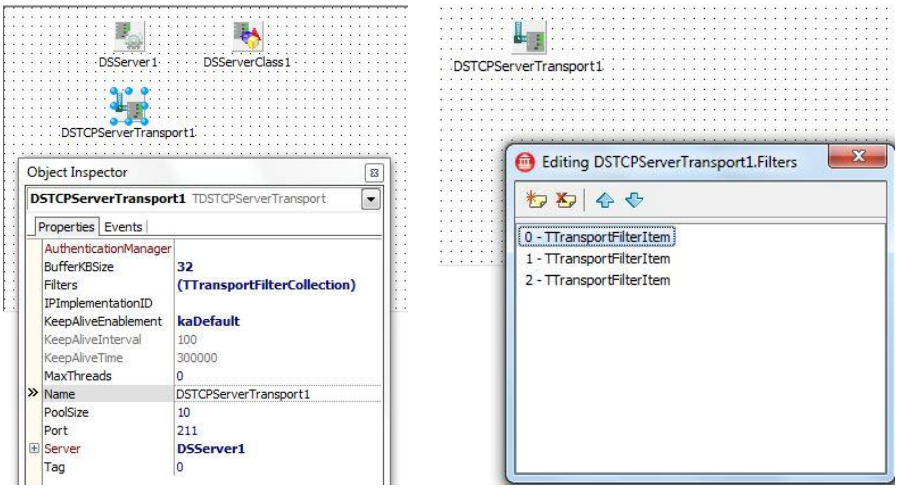
You can learn more how to build these types of DataSnap servers on the Embarcadero DocWiki at

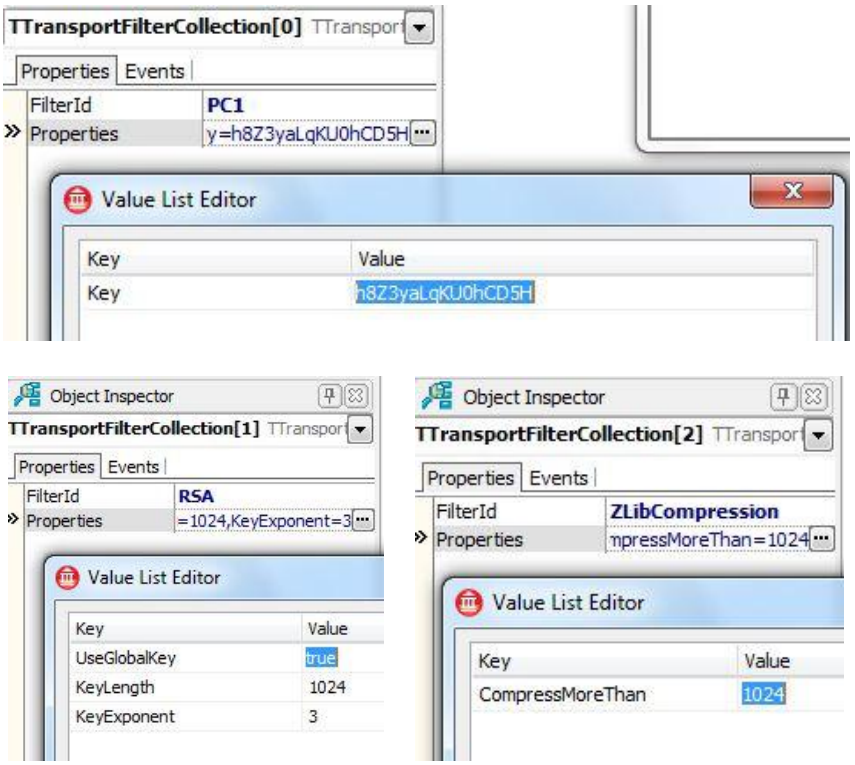
- DataSnap REST Application Wizard - http://docwiki.embarcadero.com/RADStudio/en/DataSnap_REST_Application_Wizard
- DataSnap WebBroker Application Wizard - http://docwiki.embarcadero.com/RADStudio/en/DataSnap_WebBroker_Application_Wizard
- DataSnap Server Wizard - http://docwiki.embarcadero.com/RADStudio/en/DataSnap_Server_Wizard

The Embarcadero DocWiki has complete coverage of building DataSnap applications starting at http://docwiki.embarcadero.com/RADStudio/en/Developing_DataSnap_Applications

Filtering DataSnap Byte Streams

The communication between a DataSnap client and a DataSnap server can be intercepted by a suite of filters. Each filter can perform transformations over the byte stream such as encryption and/or compression; the byte stream can be intercepted by more than one filter and such the output of one becomes the input of the next filter. The filters are attached to the byte stream at design time (or coded), by setting the Filters property of the DataSnap server transport components such as DSTCPServerTransport.TDSTCPServerTransport.





The filters are available at design time if they are present in a package registered with RAD Studio. The filter needs to be built into a package and the package needs to be installed into Delphi. Server-side design time support enables the filter to show up in the filter list editor. Client-side design-time support enables design-time connection using TSQLConnection.

There is no need to associate filters at the client side, as they are automatically instantiated based on a handshake protocol between client and server. Hence it is important that the client code registers the filters before connecting to a filtered server either by adding the unit name to the uses clause or in an early stage, such as initialization time.

You can also create your own filters for your DataSnap servers and clients. There are three things you need to do: Define a filter, Implement the filter code and Register a filter.

Defining a Filter

Any filter should extend the TTransportFilter class and the implementation needs to provide at minimum an ID that uniquely identifies it and two methods: ProcessInput and ProcessOutput.

```
public
  function ProcessInput(const Data: TBytes): TBytes; override;
  function ProcessOutput(const Data: TBytes): TBytes; override;
  function Id: UnicodeString; override;
```

The processing methods are inverse to one another: the result of one passed as input to the other produces the initial input.

Each connection instantiates a filter, so it is not necessary that the processing functions be thread safe; local variables can be used for instance, but do not assume a state-full state for an instance.

The default no-parameter constructor is used to instantiate a filter instance. Additional parameters may be needed to have the client instances compatible with the server instances. Parameter values can be exchanged between server-side filters and client-side filters. As an example, this may be necessary if a location for the encryption key needs to be passed along if one chooses to implement a symmetrical encryption filter.

All parameters are exposed as (name, string) pairs. Their names can be returned through the GetParameters method and their values can be queried or changed using GetParameterValue and SetParameterValue.

```
protected
  function GetParameters: TDBXStringArray; override;
public
  function GetParameterValue(const ParamName: UnicodeString):
    UnicodeString; override;
  function SetParameterValue(const ParamName: UnicodeString;
    const ParamValue: UnicodeString): Boolean; override;
```

In some cases, all parameters or a subset of those parameters needs to be changed at design time; their names can be provided by the GetUserParameters method.

```
protected
  function GetUserParameters: TDBXStringArray; override;
```

Note that the parameter names that are not returned by this method are not visible or editable at design time.

Implement the Filter Code

Write the code required to implement the functions defined above. You can see examples

The TTransportFilter class is defined and implemented in the source file Data.DBXTransportFilter.pas.

The two of the three Transport Filters are implemented in the following source files:

- RSA - Data.DBXRSAFilter.pas
- Zlib Compression - Data.DbxCompressionFilter.pas

Registering a filter

A filter needs to be registered with the TTransportFilterFactory singleton. The recommended way to register a filter is through the unit initialization and finalization sections, but it can be coded through an initialization phase in the user's application.

Below is the code snippet registering the compression filter available out of the box:

```
Initialization

  TTransportFilterFactory.RegisterFilter(
    TTransportCompressionFilter);

finalization
  TTransportFilterFactory.UnregisterFilter(
    TTransportCompressionFilter);
```

The Encryption Filter

The encryption filter is used to encrypt a DataSnap byte stream. The encryption filter works on the server side as well as on the client side. In the following lines the behavior of the encryption filter is described.

Server side:

- Both DSTCPServerTransport.TDSTCPServerTransport and DSHTTP.TDSHTTPService components have a Filters property.
- When selecting the filter property you can add a new filter in the dialog that comes up. In the FilterId property, you can choose PC1 or RSA.
- In case of using the PC1 encryption filter, the Properties property holds the Key value to use for the encryption. If using the RSA filter, the Properties property holds a list of three properties, UseGlobalKey, KeyLength, and KeyExponent.

Client side:

- The Driver property on the TSQLConnection has a Filters property.

In this way, when you actually run the client/server, the communication between them will be encrypted. There are some things to take into account while using the encryption filter with DataSnap client/server applications:

- The data encryption is not available with thin clients.
- If the server has an encryption filter but the client does not, then the client will automatically add the filter.
- If the client has an encryption filter but the server does not, then the client will drop its filter and will not use it.

The Compression Filter

The compression filter is based on ZLib and provides compression capabilities for DataSnap byte streams. The compression filter works both on the server side and the client side.

The DSTCPServerTransport.TDSTCPServerTransport and DSHTTP.TDSHTTPService components have a Filters property. This is where you add a new filter setting the FilterId property to ZLibCompression. You can also specify properties for the compression filter by setting the Properties property.

The Delphi Labs: DataSnap series – Episode 6: "DataSnap Transport Filters" white paper shows you how to use Filters in your Delphi DataSnap applications - http://edn.embarcadero.com/article/41293

Using Web Services in your Windows and Mac Applications

Web Service applications are server implementations that do not require clients to use a specific platform or programming language. These applications define interfaces in a language-neutral document, and they allow multiple communication mechanisms.

Web Services were first designed to work using Simple Object Access Protocol (SOAP) - http://en.wikipedia.org/wiki/SOAP. SOAP is a standard lightweight protocol for exchanging information in a decentralized, distributed environment. SOAP uses XML to encode remote procedure calls and typically uses HTTP as a communications protocol.

SOAP Web Service applications use a Web Service Definition Language (WSDL), http://www.w3.org/TR/wsdl, document to publish information on interfaces that are available and how to call them. On the server side, your application can publish a WSDL document that describes your Web Service. On the client side, a wizard or command-line utility can import a published WSDL document, providing you with the interface definitions and connection information you need. If you already have a WSDL document that describes the Web service you want to implement, you can generate the server-side code when you import the WSDL document.

Another Web Service software architecture, REST based Web Services, has emerged in recent years to become the dominant World Wide Web standard. REST stands for Representational State Transfer and was created to work with HTTP v1.1. REST got its start as Roy Fielding's University of California Doctoral Dissertation, "Architectural Styles and the Design of Network-based Software Architectures". You can find Fielding's dissertation at http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (Fielding, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000). Chapter 5 of the dissertation introduces REST - http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

Conforming to REST constraints is referred to as being RESTful. A RESTful web service is implemented using HTTP (hypertext transfer protocol) and the principles of REST. REST assumes that the server is able to serve four types of requests: GET, POST, PUT, and DELETE. These operations represent Retrieve, Update, Insert, and Delete data operations. These operations are assimilated with the server methods through a mapping protocol. REST assumes that each server method can be invoked as one of the operations above through a dispatch mechanism that assumes a mapping between the URI (Uniform Resource Identifier) path, method name, and parameters.

For additional information about building Web Services, take a look at the following web sites and videos:

- Embarcadero DocWiki article about REST - http://docwiki.embarcadero.com/RADStudio/en/REST
- DataSnap supports building REST servers - http://docwiki.embarcadero.com/RADStudio/en/DataSnap_REST_Application_Wizard
- W3C SOAP v1.1 specification - http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
- Writing Servers that support Web Services - http://docwiki.embarcadero.com/RADStudio/en/Writing_Servers_that_Support_Web_Services
- Writing Clients for Web Services - http://docwiki.embarcadero.com/RADStudio/en/Writing_Clients_for_Web_Services
- Building and Consuming Web Services with Delphi 2009 - http://blogs.embarcadero.com/pawelglowacki/2008/12/18/38624/

Creating a Web Service Application and a Windows and Mac Client application that uses the services

In this section we'll build a Simple Calculator SOAP Web Server and build a FireMonkey client application that calls the Web Service's methods and displays the results. In this example, I will follow similar building web service steps that Pawel Glowacki documented in his Delphi 2009 based blog post at http://blogs.embarcadero.com/pawelglowacki/2008/12/18/38624/. In Pawel's example, he built a web service based CGI application that required you to have Microsoft's IIS (Internet Information Server). For my example, I will build a web services console application that can run as a standalone web server application.

There are many types of SOAP Web Service application that you can create with Delphi and C++Builder. The "New SOAP Server Application" wizard lets you build:

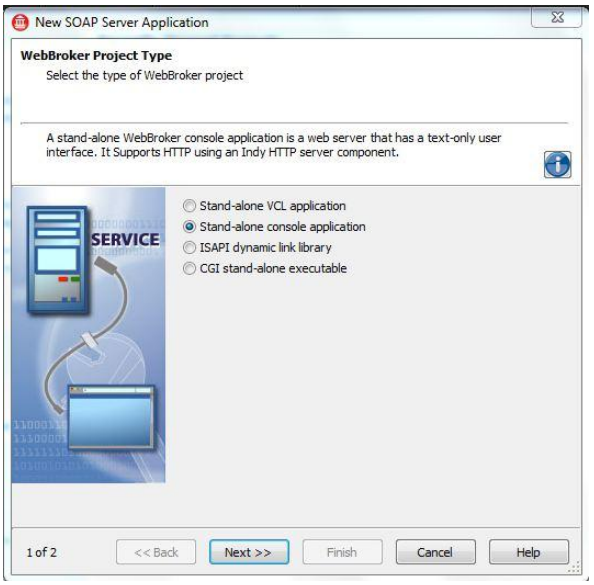
- Stand-alone VCL application - An Internet Direct (Indy or Indy.Sockets) application that uses VCL forms.
- Stand-alone console application - An Internet Direct (Indy or Indy.Sockets) console application.
- ISAPI dynamic link library - ISAPI Web server applications are DLLs that are loaded by the Web server. Client request information is passed to the DLL as a structure. Each request message is handled in a separate execution thread.
- CGI stand-alone executable - A CGI stand-alone Web server application is a console application that receives client request information on standard input and passes the results back to the server on standard output. Each request message is handled by a separate instance of the application.

As previously noted, you can also build DataSnap REST server applications that implement RESTful web services using TCP, HTTP and HTTPs. You can also use the Indy server components to create your own server side services.

Step 1 – Create the simple calculator SOAP server application

To start building the Simple Calculator Web Services application we'll start by using the "New Soap Server Application" wizard. Use **File > New > Other > Delphi Projects > Web Services > SOAP Server Application** or **File > New > Other > C++Builder Projects > Web Services > SOAP Server Application** to bring up wizard.

This is the first of two wizard dialog boxes that will help create the type of server your Web Service application will work with.



To keep things simple, we'll choose Stand-alone console application. You could also create your own starting project (not using the wizard) and add Web Server and Web Service components and code to create a FireMonkey based server application. After making the selection click the "Next" button.

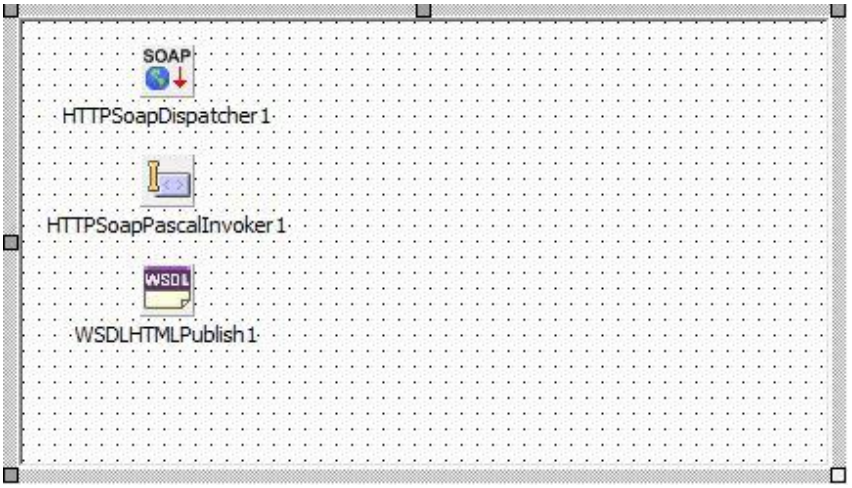
Page 2 of the wizard lets you choose the HTTP port number for your SOAP server application. You can input a port and click the "Test Port" button to make sure that port is available. You can also click the "Find Open Port" button and the wizard will find an available port for your application to use. If you want to use secure HTTPS you can click the checkbox. For our simple calculator we will choose port 8085 and then click the "Finish" button.

The wizard generates a new Web server application that includes a Web module which contains three components:

- An invoker component (Soap.SOAPHTTPPasInv.THTTPSoapPascalInvoker). The invoker converts between SOAP messages and the methods of any registered invokable interfaces in your Web Service application.

- A dispatcher component (Soap.WebBrokerSOAP.THTTPSoapDispatcher). The dispatcher automatically responds to incoming SOAP messages and forwards them to the invoker. You can use its WebDispatch property to identify the HTTP request messages to which your application responds. This involves setting the PathInfo property to indicate the path portion of any URL directed to your application, and the MethodType property to indicate the method header for request messages.

- A WSDL publisher (Soap.WSDLPub.TWSDLHTMLPublish). The WSDL publisher publishes a WSDL document that describes your interfaces and how to call them. The WSDL document tells clients that how to call on your Web Service application. For details on using the WSDL publisher, see Generating WSDL Documents for a Web Service Application.



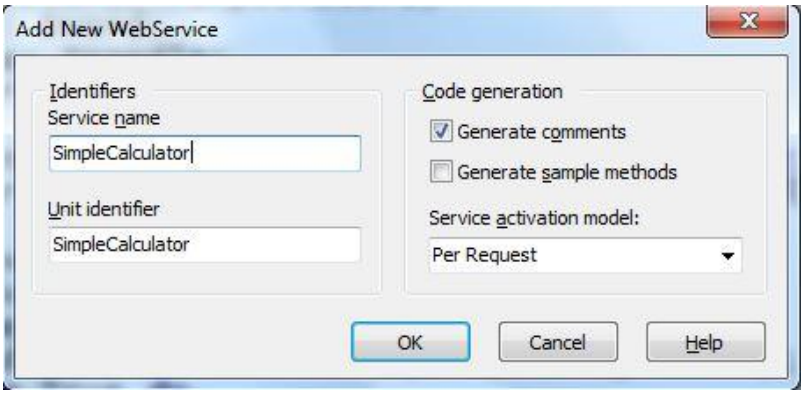
The SOAP dispatcher and WSDL publisher are auto-dispatching components. This means they automatically register themselves with the Web module so that it forwards any incoming requests addressed using the path information they specify in their WebDispatch properties. If you right-click on the Web module, you can see that in addition to these auto-dispatching components, it has a single Web action item named DefaultHandler.

DefaultHandler is the default action item. That is, if the Web module receives a request for which it can't find a handler (can't match the path information), it forwards that message to the default action item. DefaultHandler generates a Web page that describes your Web Service. To change the default action, edit this action item's OnAction event handler.

Step 2 – Create the starting Web Service Interface

After the project is created, the next step is to create the interfaces and implementations for the methods your Web Service will provide. You will see the following dialog box appear:

Click the Yes button to start creating your Web Service interfaces. You can also choose to create the interfaces later on.



Type SimpleCalculator for your service name. The unit name will also echoed but you can create any source file unit name you want for your service interfaces and implementations.

The Add New Web Service wizard lets you specify the name of the invokable interface you want to expose to clients, and generates the code to declare and register the interface and its implementation class. The wizard has options to also generate comments and sample methods and additional type definitions, to help you get started in editing the generated files. Since we are going to create calculator methods, keep the "Generate comments" check box checked and don't check the "Generate sample methods" check box.

You can choose the Service activation model in the dialog. The choices are:

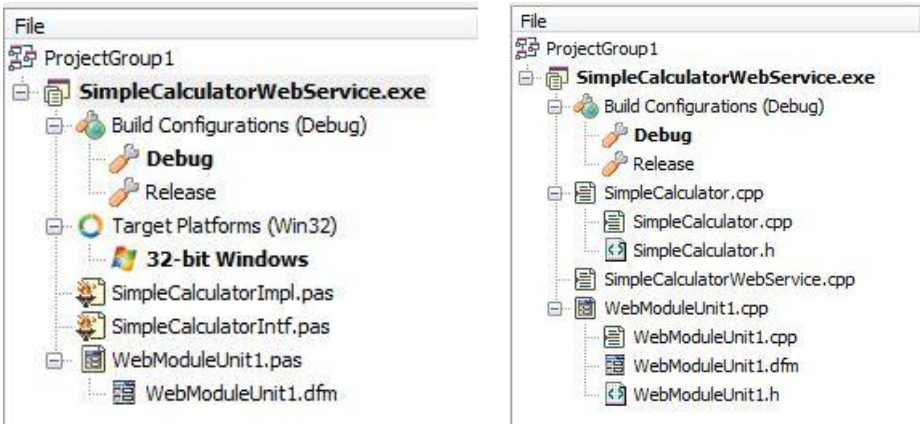
- Per Request creates a new instance of your implementation class in response to each request it receives. That instance is freed after the request is handled.

- Global Object creates a single instance of your implementation class, which is used to handle all requests.

To keep things simple, choose the "Per Request" Service activation model.

Save the Web Service server application project – **File > Save All.** Click OK to keep the names for the web service implementation files and the web module. Save the project name a s "SimpleCalculatorWebService".

The Project Manager view for your Delphi or C++ Web Service application will look like the following:



Step 3 – Define and implement your Web Service methods

To complete the Web Service application we need to define and implement the methods that client applications will use. For our simple calculator, we'll create the addition and subtraction methods. A follow on exercise for you will be to define and implement the multiply and divide methods to complete the simple calculator web service.

For Delphi,

a) Add the following lines in the SimpleCalculatorIntf.pas file's interface section inside the public area of the ISimpleCalculator interface declaration:

```
function Add(a,b: integer): integer; stdcall;
function Subtract(a,b: integer): integer; stdcall;
```

b) Add the following code in the SImpleCalculatorImpl.pas file's interface and implementation sections:

```
{ TSimpleCalculator }
TSimpleCalculator = class(TInvokableClass,
    ISimpleCalculator)
  public
    function Add(a,b: integer): integer; stdcall;
    function Subtract(a,b: integer): integer; stdcall;
  end;
```

```
function TSimpleCalculator.Add(a,b: integer): integer; stdcall;
begin
  Result := a+b;
end;

function TSimpleCalculator.Subtract(a,b: integer): integer;
stdcall;
begin
  Result := a-b;
end;
```

For C++,

a) Add the following lines in the SimpleCalculator.h file's area of the ISimpleCalculator interface declaration:

```
public:
  virtual int Add(int a, int b) = 0;
  virtual int Subtract(int a, int b) = 0;
```

b) Add the following declarations in the SimpleCalculator.cpp file's public section of the TSimpleCalculatorImpl class:

```
public:
  int Add(int a, int b);
  int Subtract(int a, int b);
```

c) Add the following function implementations in the SimpleCalculator.cpp file to implement the calculator methods:

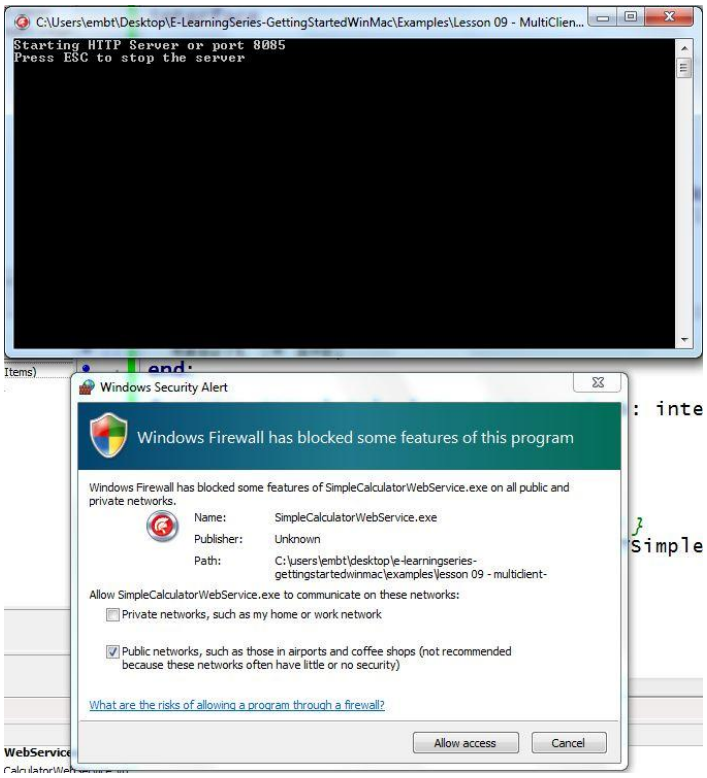
```
int TSimpleCalculatorImpl::Add(int a, int b) {
  return a + b;
}

int TSimpleCalculatorImpl::Subtract(int a, int b){
  return a-b;
}
```

Your SOAP server is now implemented and ready to run.

Choose **File > Save All** to save your changes.

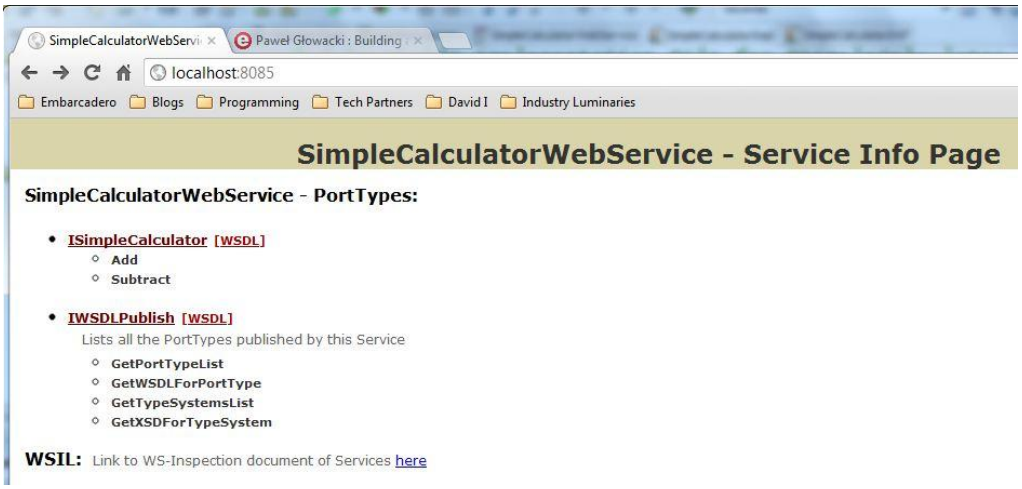
Step 4 – Run your SOAP Web Server Application.

Hit Shift-Control-F9 or choose Run without Debugging to start your Web Server console application. You will see the following console window appear along with (the first time you run it) a Windows FireWall warning for the applications request to use a TCP/IP port:



Click the "Allow Access" button for the SimpleCalculatorWebService.exe application. Your Web Service is now ready to receive method calls on port 8085 or whatever port you chose.

To test the Web Service you can point your browser at the server and port where the application is running. If it is your development machine you can use http://localhost:8085/ to access the interfaces of your Web Service.



This browser result page shows you the Web Services defined in your Web Server application and also the WSDL published interfaces. You can point your browser at other Internet web services applications and see similar results. You can right mouse-click on the ISimpleCalculator [WSDL] link to place the URL on the clipboard for use a later step using the WSDL Import wizard.

Step 5 – Create a FireMonkey Client Application that will consume the Web Service

Now that the Web Services application is running, we can create a FireMonkey client application (HD or 3D) that will provide the UI and call the remote methods to use the Simple Calculator.

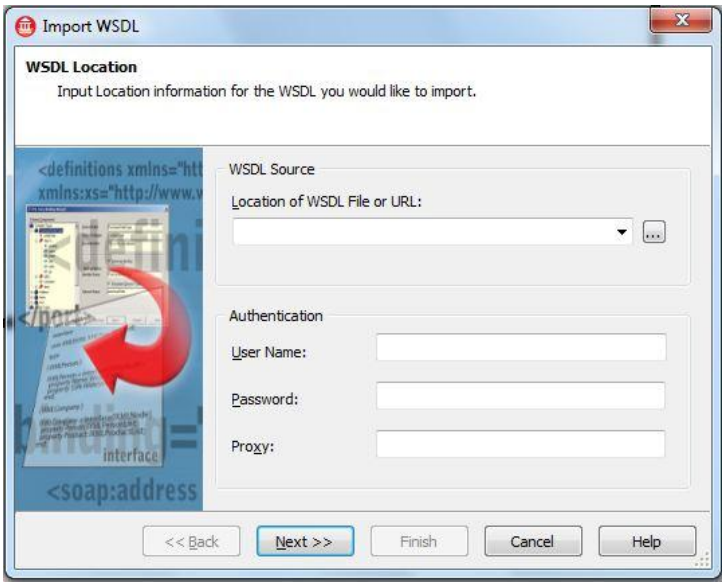
Start building a FireMonkey HD application. In the Project Manager view - right mouse-clicking or click the Add New Project speed button and choose from the project categories **FireMonkey HD Application – Delphi** or **FireMonkey HD Application – C++Builder**. This will add a new project to go with the Server application.

Save the client project and project group by clicking "Save All". Name the Client project "DelphiFMClientApplication" or "CppFMClientApplication". Name the Project Group "DelphiCalcWebServicesProjectGroup" or "CppCalcWebServicesProjectGroup" (or whatever name you choose).

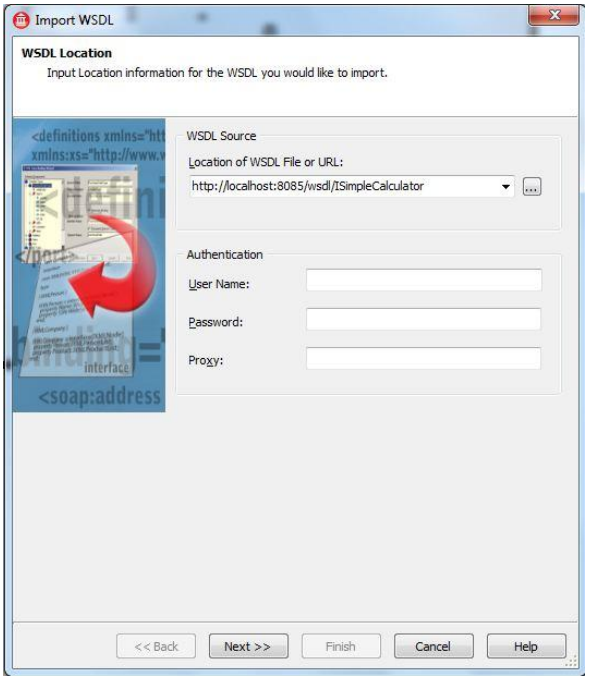
Step 6 – Use the WSDL importer to create an Web Services interface unit for your client application

Your client application will need the interfaces for the Web Services methods. To create the interface file(s) the client needs we'll use the WSDL importer wizard. Remember to have your Web Server application running.

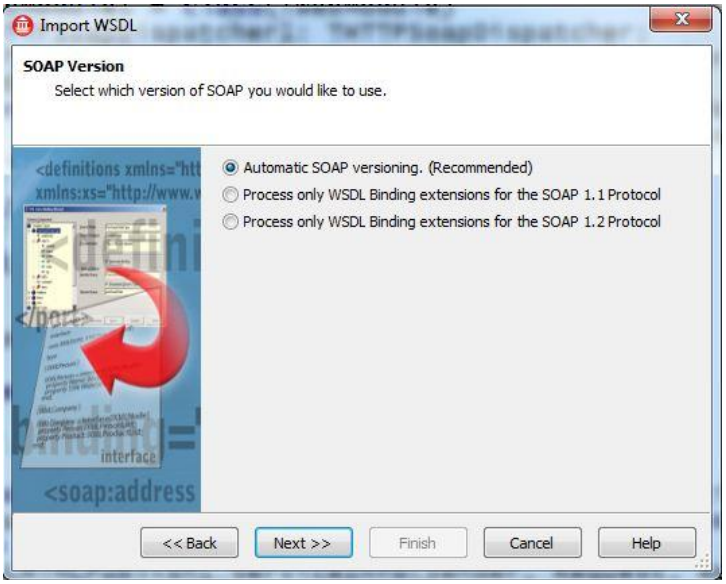
Choose **File > New > Other…** and from the Delphi Projects or C++ Builder Projects Web Services Category, choose the WSDL Importer wizard.



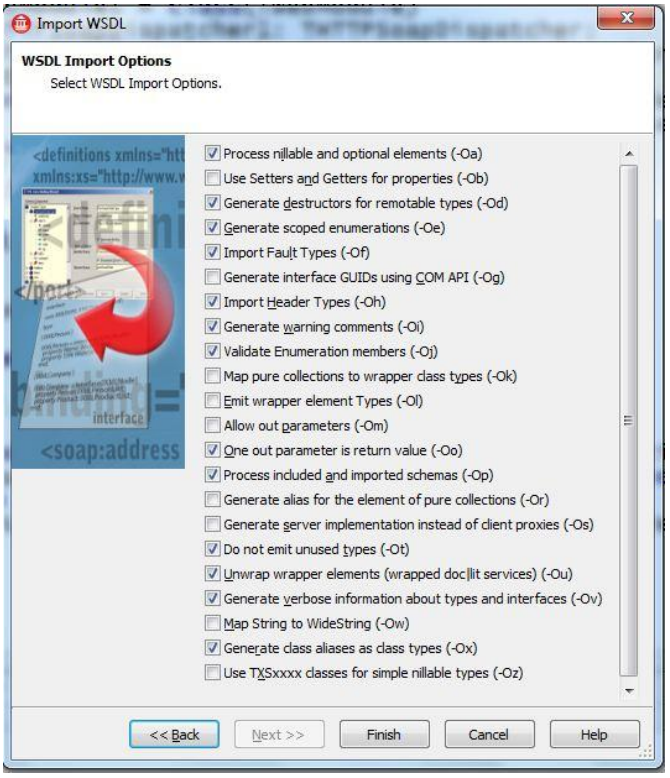
Type or paste the URL location for your Web Services application. In the simple calculator case this is "http://localhost:8085/wsdl/ISimpleCalculator".



If the web service requires login authentication, you can enter that information in the provided boxes. For our simple calculator web service, just click the "Next" button. You will then see the second page of the WSDL Import Wizard where you can select which WDSL binding extensions to use.



The default choice is to let the WSDL Import Wizard get the version information from the SOAP Server and generate the right interfaces for you. Leave "Automatic SOAP versioning" selected and click the "Next" button.



The third and final dialog in the WSDL Import Wizard allows you to choose from the many different options. Use this wizard page to configure the way the wizard generates code to represent definitions in a WSDL document. You should typically use the default options, because the defaults provide the safest way to import WSDL documents. Information about each of the options is available by clicking the "Help" button and on the Embarcadero DocWiki at http://docwiki.embarcadero.com/RADStudio/en/Import_WSDL_Wizard.

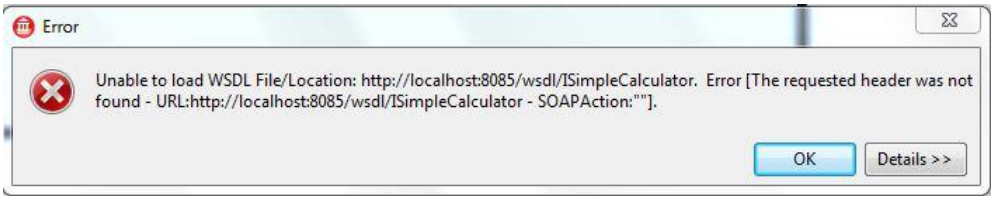
We'll leave all of the default options and click the "Finish" button.

The following error messages can be generated by the Import WSDL wizard:

- Unable to load WSDL File/Location: <wrongURL>. Error [Empty document] - This error message typically indicates an invalid URL. That is, a URL that does not return any content.
- Unable to load WSDL File/Location: <url>. Error [Whitespace is not allowed at this location] - This error indicates a URL that returned HTML or text content (i.e. a URL that did not return XML content)
- Unable to load WSDL File/Location: <url>. Error [End tag 'ul' does not match the start of tag 'p'] - This error also indicates a URL that returned HTML content. This message is common if you forgot the '?wsdl' query string commonly used by some WebServices. For example, if you use http://<domain>.com/service.asmx instead of http://<domain>.com/service.asmx?wsdl for a .NET WebService.
- *Error*: '<url>' - Missing <definition> node of namespace http://schemas.xmlsoap.org/wsdl/. - This error indicates that the URL returned XML content but that latter did not contain the root

<definition> element expected for a WSDL file. This will happen if you import the schema used by a WSDL instead of the WSDL document itself, for example.

If the Web Service is not running, you will most likely see the following error message when you click the "Finish" button on the WSDL Import Wizard final dialog box.

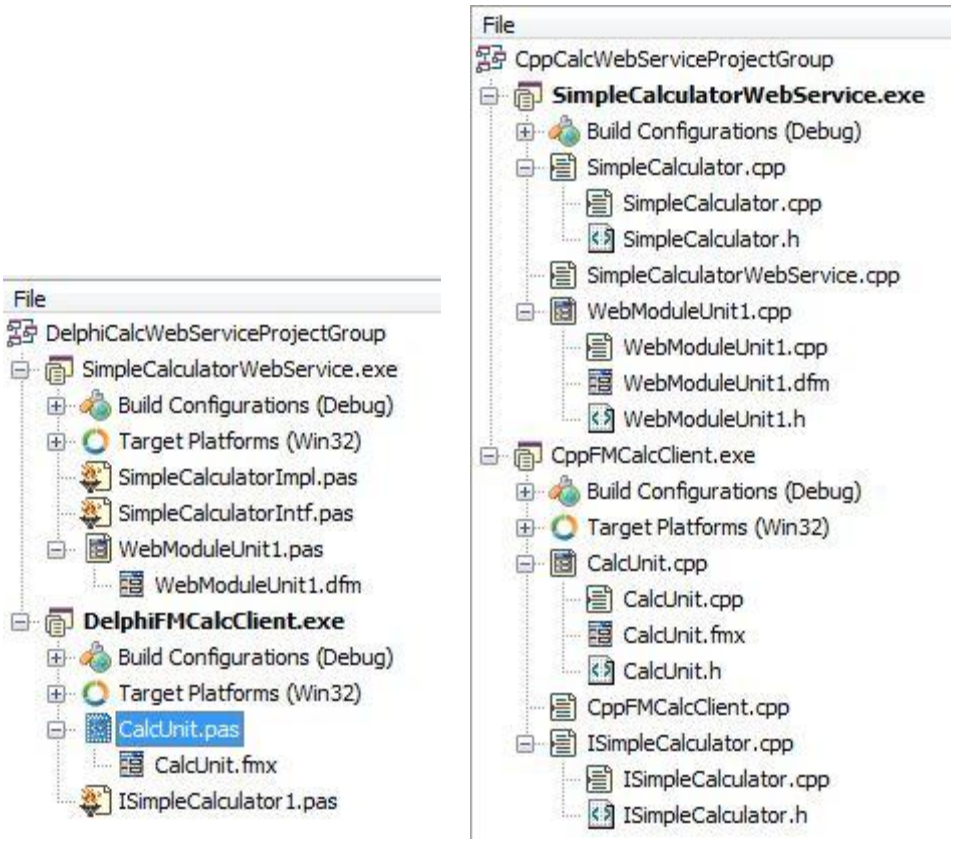


If any error happens, make sure your Web Service application is running and restart the WSDL Import Wizard and step through the dialogs. A simple way to troubleshoot your URL is to view the WSDL URL in your browser. You should see the service info page that I showed you earlier.

If all is good, then a source code file (and a header file for C++Builder) will be generated containing the declaration and implementation of the interfaces for the Simple Calculator that you can use in your client application. You can also use the WSDL Import Wizard for your server application for the cases when your Web Service also uses other Web Services.

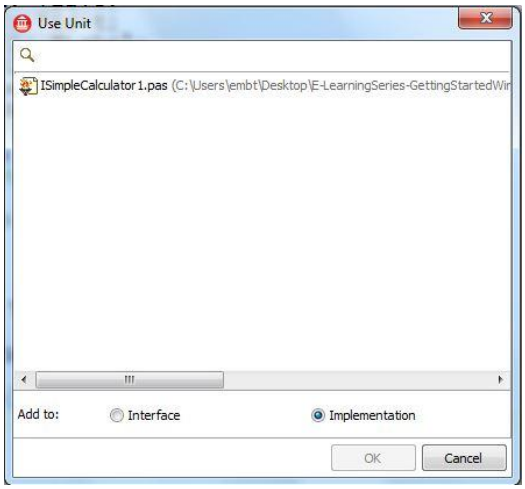
For Delphi, the wizard generates the ISimpleCalculator1.pas file. For C++, the wizard generates the ISimpleCalculator.cpp and ISimpleCalculator.h files.

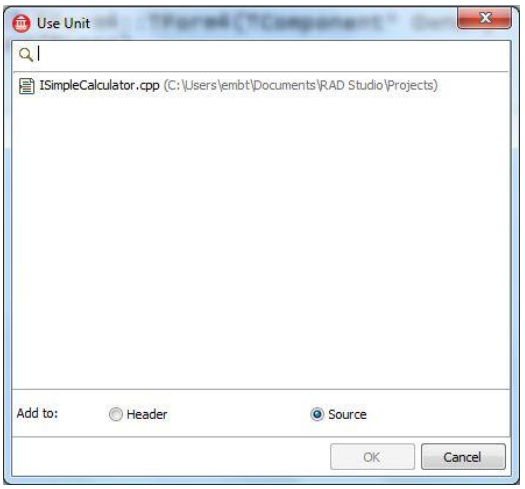
Save the project group (server and client) using **File > Save All**. You can leave the filenames as they were created by the wizard. Your Project Manager view should now look something like the following images for your Delphi or C++ project group.



Step 7 – Create the UI for the Client Application and use the Web Service Methods

In order to start using the remote methods in the client application, we need to make sure that we have access to the file(s) generated by the WSDL Import Wizard. Make sure your client application is the active project in the group and select the CalcUnit.pas or CalcUnit.cpp file (or the client form file that you created) and use **File > Use Unit** to add the ISimpleCalculator file(s) to your Delphi unit's implementation section or your C++ unit's source code.

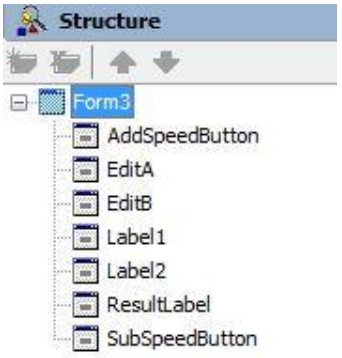


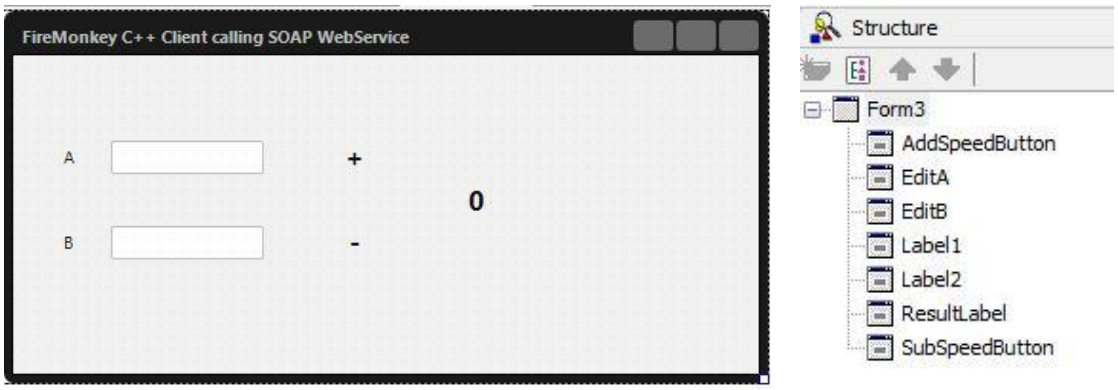



```
// Delphi
implementation
{$R *.fmx}
uses ISimpleCalculator1;

// C++
#include "ISimpleCalculator.h"
```

Now that we have access to the Web Servers remote methods, we can build the User Interface that will use them. The Simple Calculator Web Service has Add and Subtract methods. To complete the UI, we'll need to add two edit boxes for input, two TLabel components to identify the edit boxes, two TSpeedButton (or TButton) components to call the Add and Subtract methods, and a TLabel to display the method result.

Your client form and structure view should look something like the following:





Finally, we can add event handlers for the Add (+) and Subtract (-) speed buttons to take the contents of the edit boxes, call the Web Service methods and display the result in the label (above with the 0 text string). Double click on each of the speed buttons (or use the Object Inspector's events tab and select the OnClick event) to create the skeleton event handler and then add the following code.

```
//Delphi
procedure TForm3.AddSpeedButtonClick(Sender: TObject);
var
  a,b,c: integer;
begin
  a := StrToInt(EditA.Text);
  b := StrToInt(EditB.Text);
  c := GetISimpleCalculator.Add(a,b);
  ResultLabel.Text := IntToStr(c);
end;

procedure TForm3.SubSpeedButtonClick(Sender: TObject);
var
  a,b,c: integer;
begin
  a := StrToInt(EditA.Text);
  b := StrToInt(EditB.Text);
  c := GetISimpleCalculator.Subtract(a,b);
  ResultLabel.Text := IntToStr(c);
end;

// C++
void __fastcall TForm3::AddSpeedButtonClick(
    TObject *Sender) {
  int a = StrToInt(EditA->Text);
  int b = StrToInt(EditB->Text);
  int c =
    NS_ISimpleCalculator::GetISimpleCalculator()->Add(a,b);
  ResultLabel->Text = IntToStr(c);
}
//---------------------------------------------------------------
------------
void __fastcall TForm3::SubSpeedButtonClick(
    TObject *Sender) {
  int a = StrToInt(EditA->Text);
  int b = StrToInt(EditB->Text);
  int c =
    NS_ISimpleCalculator::GetISimpleCalculator()->Subtract(a,b);
```

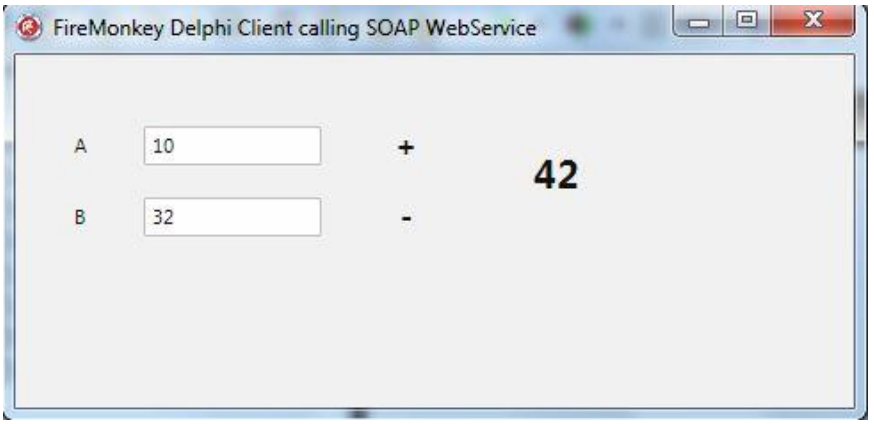
```
  ResultLabel->Text = IntToStr(c);
}
```

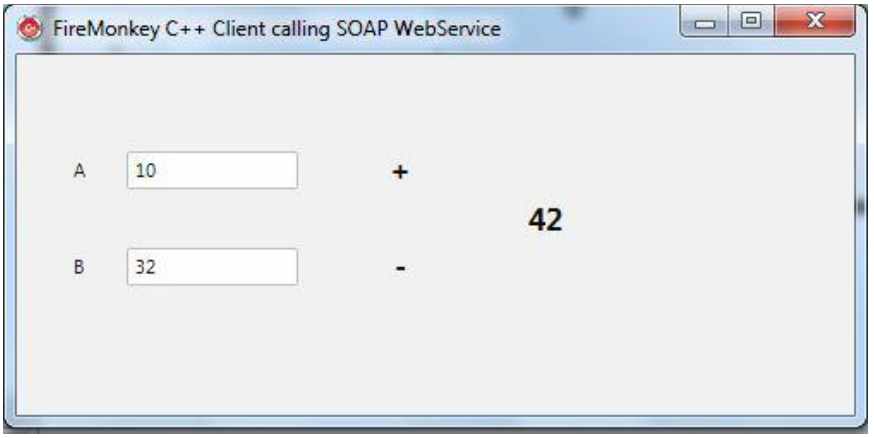
Save your work with **File > Save All**

Step 8 – Compile and Run the Client Application

With your Web Service application running and the client application activated in your project group, hit F9 or choose Run or Run without Debugging. If all is well in our source code, the client application will compile and run.

Enter numbers in the two edit boxes and click the Add and Subtract buttons to see the result displayed.



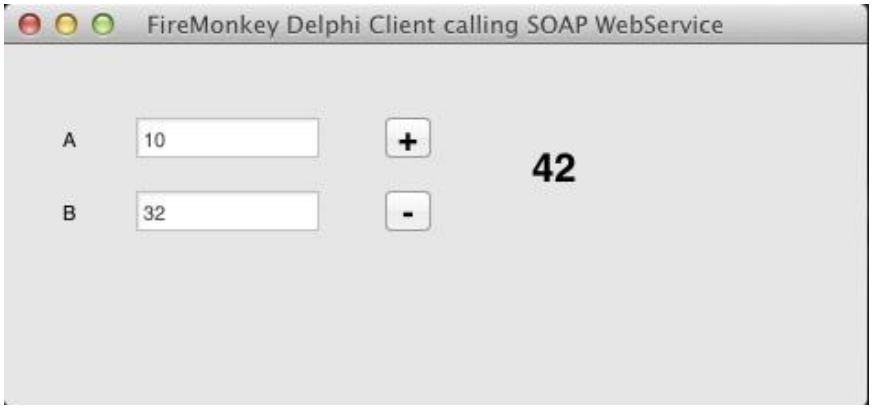


If you want to run the client application on the Mac, modify the constant strings in the ISimpleCalculator1.pas (Delphi) and ISimpleCalculator.cpp (C++) unit to point to the TCP/IP and port address where your Web Service is running (instead of using localhost).

```
// Delphi
// Filename: ISimpleCalculator1.pas
function GetISimpleCalculator(UseWSDL: Boolean;
  Addr: string; HTTPRIO: THTTPRIO): ISimpleCalculator;
```

```
const
  defWSDL = 'http://localhost:8085/wsdl/ISimpleCalculator';
  defURL  = 'http://localhost:8085/soap/ISimpleCalculator';

// C++
// Filename: ISimpleCalculator.cpp
_di_ISimpleCalculator GetISimpleCalculator(
  bool useWSDL, System::String addr,
  Soaphttpclient::THTTPRIO* HTTPRIO) {
  static const char* defWSDL=
    "http://localhost:8085/wsdl/ISimpleCalculator";
  static const char* defURL =
    "http://localhost:8085/soap/ISimpleCalculator";
```

Save the client project, select the Target Platform to OSX and run the client and hit F9 to run the Mac client.

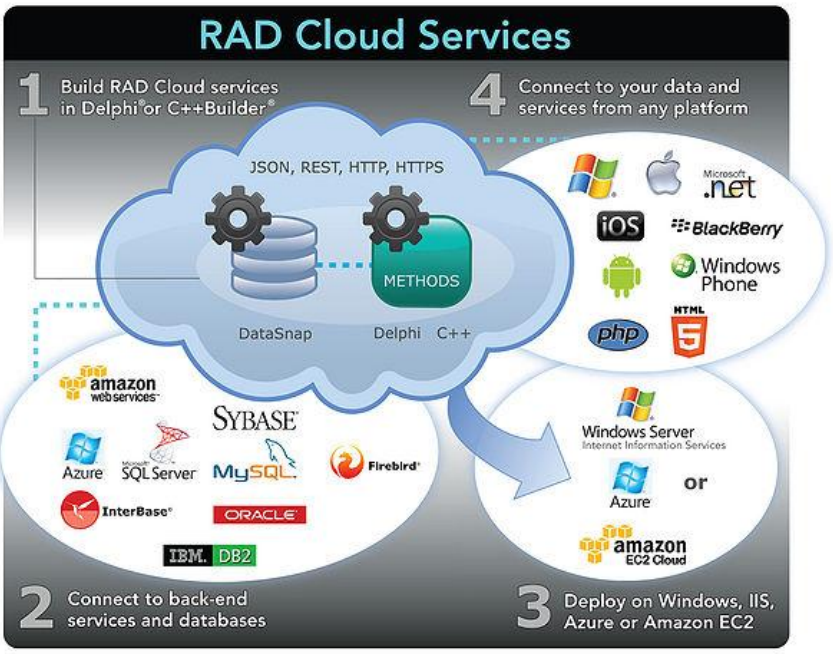


That's It! You've created your first Web Services server and client application for Windows and Mac. You can also create Windows and Mac client applications that can use external Internet Web Services that Amazon, eBay and others provide. Try using the WSDL Import Wizard on those services and build a client that uses their interfaces.

You will find additional information about Web Services on the Embarcadero DocWiki at:

- http://docwiki.embarcadero.com/RADStudio/en/Web_Services_Overview
- http://docwiki.embarcadero.com/RADStudio/en/Using_Web_Services
- http://docwiki.embarcadero.com/RADStudio/en/Import_WSDL_Wizard
- http://docwiki.embarcadero.com/RADStudio/en/Writing_Servers_that_Support_Web_Services
- http://docwiki.embarcadero.com/RADStudio/en/Writing_Clients_for_Web_Services
- http://docwiki.embarcadero.com/RADStudio/en/Developing_Web_Services_with_Win32_Applications
- http://docwiki.embarcadero.com/RADStudio/en/Building_a_Hello_World_Web_Services_Application

Using Cloud Storage and Services in your Windows and Mac Applications



Beside DataSnap, which is the technology that allows you to build multi-tier applications, Rad Studio XE2 provides Cloud components, which allow you to easily use cloud services from Amazon and Microsoft Azure. RAD Studio provides a framework that allows you to build cloud services and easily connect to your back-end services and databases.

With RAD Studio's RAD Cloud deployment, you can move your data and services to the Cloud, making your applications accessible from virtually any platform or device from anywhere in the world.

RAD Studio Cloud Services

The RAD Studio CloudAPI unit:

- Holds the common functionality for the Amazon and Azure APIs, and any other Cloud service that could be implemented, which uses a REST API and the same (or a very similar) authentication mechanism.
- Has SHA1 and SHA256 authentication implementations. Because of this, OpenSSL libraries are required when using any API that extends this unit.
- Contains useful classes that are common across multiple APIs, such as TCloudResponseInfo and TCloudTableRow.
- Contains the TCloudService class, which is the base service class which the Amazon and Azure API service classes extend.

Azure Cloud Service

The AzureAPI unit is a redesign of the original DSAzure API shipped with RAD Studio. The API is now more intuitive. It is easier to find the function you want to call, and it is obvious how to get the result of the API call. It should also now be thread safe, so you do not need to make a new instance of the service class in new threads, if you do not want to. Also, functions have been introduced to do the XML parsing of results for you. There are now classes and lists that wrap various XML responses. You can still call the versions of these functions that return XML, if you want to do the parsing yourself. The service classes are: TAzureBlobService, TAzureQueueService, TAzureTableService. The designer component for creating a connection info instance is TAzureConnectionInfo.

Amazon Cloud Service

The AmazonAPI unit is completely new to RAD Studio. It includes support for the Amazon Simple Queue Service (SQS), Amazon Simple Storage Service (S3) and Amazon SimpleDB service. These are very similar to their comparable services offered by Amazon. You use this API in the same way you use the AzureAPI, but each of the services has subtle differences. You can see the Amazon REST API documentation and Azure REST API documentation for a better idea of the similarities and differences. Like the Azure API, the Amazon API has functions that will do the XML parsing for you, unless you want to implement that yourself. The service classes are: TAmazonTableService, TAmazonQueueService, TAmazonStorageService. The designer component for creating a connection info instance is TAmazonConnectionInfo.

Building your first Cloud base Windows and Mac Application

Since this Lesson 9 course book is already over 70 pages (and I am late to get this into your hands and also going on summer vacation), I am going to provide you with links to the Embarcadero DocWiki for now and finish a step by step tutorial in the coming weeks:

- http://docwiki.embarcadero.com/RADStudio/en/Developing_Cloud_Applications
- http://docwiki.embarcadero.com/RADStudio/en/Cloud_Computing_with_DataSnap
- http://docwiki.embarcadero.com/RADStudio/en/Azure_and_Cloud_Computing_with_DataSnap
- http://docwiki.embarcadero.com/Libraries/en/Data.Cloud.CloudAPI

- http://docwiki.embarcadero.com/Libraries/en/Data.Cloud.AzureAPI
- http://docwiki.embarcadero.com/Libraries/en/Data.Cloud.AmazonAPI

RAD Studio ships with a CloudAPI example for Delphi and the Embarcadero DocWiki has additional information.

- http://docwiki.embarcadero.com/CodeExamples/en/DataSnap.Cloud_Explorer_Sample
- C:\Users\Public\Documents\RAD Studio\9.0\Samples\Delphi\CloudAPI\CloudExplorer

We also have a video showing you how to use Amazon and Azure Cloud Services in your Delphi XE2 and C++Builder XE2 Applications:

- http://edn.embarcadero.com/article/41902

Here are some links to information about Microsoft and Amazon cloud services:

- http://msdn.microsoft.com/en-us/library/dd163896.aspx
- http://aws.amazon.com/documentation/sqs/
- http://aws.amazon.com/documentation/s3/
- http://aws.amazon.com/documentation/simpledb/

Deploy your application to the Cloud

While we focused this section on using cloud storage and services in your Windows and Mac applications, you can also deploy your Windows applications to the cloud using the Deployment Manager.

You can deploy your application to an Amazon EC2 server just as you would deploy to any other computer. You'll need to install the Windows Platform Assistant on the EC2 server and set up a local profile in RAD Studio with the machine name and port of the server. After that, you will be able to use the deployment manager to deploy your application (and any other required files) to the EC2 server. If the remote debugger is installed on the EC2 server, then it can be used to debug the deployed applications.

Summary, Looking Beyond, To Do Items, Resources, Q&A and the Quiz

In Lesson 9 you learned how to build multi-client, multi-platform and multi-tier applications for Windows and Mac. There are many additional aspects to this topic that we did not have time to cover. But, at least you learned how to quickly build Windows and Mac client applications that work with DataSnap servers, Cloud services and Web Services. There is much more to explore and I have included links to articles and videos that will help you learn more.

This is the last scheduled lesson in the "Getting Started with Windows and Mac Development". We could cover many additional getting started topics beyond the 9 lessons already presented. I hope to

add additional lessons in the future, so, please continue to send me email messages for topics you want to learn more about.

Where do you go from here? Start building your own Windows and Mac applications with Delphi, C++ and FireMonkey. Send me emails (davidi@embarcadero.com) telling me about all the cool applications you are building for Windows and Mac. If you have any questions, suggestions, course book corrections or need additional links to resources, examples, videos etc – just let me know.

Good Luck and Keep on Programming!!!

To Do Items

Add the Multiply and Divide methods to your Web Services server application and update the client with Speed Buttons (or Buttons) to call the new methods.

Compile and run the DataSnap and Web Services client applications on Windows (Win32/64 for Delphi, Win32 for C++) and Mac (OSX32 for Delphi and C++). If you have time, try mixing Delphi and C++ clients for both the DataSnap and Web Services server applications. See for yourself that the Delphi client can work with the C++ server application and the C++ client can work with the Delphi server application.

Use the WSDL Import Wizard with other external Internet web services like Amazon, eBay to create client applications that use their interfaces.

Links to Additional Resources

- Getting Started Course landing page - http://www.embarcadero.com/firemonkey/firemonkey-e-learning-series
- Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000. - http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
- Pawel Glowacki's Delphi Labs – Series 1: DataSnap – 11 episodes - http://www.embarcadero.com/rad-in-action/delphi-labs
- The TIndex.net directory of FireMonkey resources - http://www.tindex.net/FireMonkey.html
- Building Web Services the REST Way by Roger Costello - http://www.xfront.com/REST-Web-Services.html

Q&A:

Here are some of the answers for the questions I've received (so far) for this lesson. I will continue to update this Course Book during and after course.

If you have any additional questions – send me an email - davidi@embarcadero.com

Self Check Quiz

1. Delphi and C++ DataSnap server applications can run on which operating system?

A: Windows
B) Mac
C) Linux

2. Delphi and C++Builder support which of the following industry standard Web Service architectures?

a) SOAP
b) REST
c) TOAST
d) WSDL

3. Which RAD Studio component provides the primary connection from a client application to a Delphi or C++Builder DataSnap server?

a) THTTPRIO
b) TSQLConnection
c) TDSServer
d) TDataSource

4. Which Cloud Service is not directly supported by RAD Studio XE2?

a) Amazon Web Services/EC2
b) R.A.I.N
c) Microsoft Windows Azure

Answers to the Self Check Quiz:

1a, 2a&b, 3b, 4b