Tutorial: Using a REST DataSnap Server with an Application and FireDAC

Contents

Follow this tutorial to build a multi-tier database application with DataSnap framework. The server is a web application that accesses data from an InterBase database. The client is a FireMonkey desktop application that includes a DataSnap REST Client Module. The client uses the HTTP protocol to connect to the server and to exchange JSON data through REST interfaces.

FireDAC JSON Reflection support was introduced in RAD Studio XE5 Update 2.

This tutorial uses the following technologies:

- Database: Interbase
- Database access framework: FireDAC
- Data format: JSON
- Client/Server architecture: REST
- Communication protocol: HTTP
- Client-side in-memory dataset: FireDAC **TFDMemTable**
- UI technology: Visual LiveBindings

Creating the DataSnap REST Application

1. Create a new project:
   - Choose **File > New > Other** from the main menu.
   - Go to the **DataSnap Server** node in the left column, under the C++Builder Projects or the Delphi Projects node.
   - Select **DataSnap REST Application** and press OK.
   - Specify the type of application:
     1. Select **Stand-alone application** and press **Next** .
     2. Choose one of the options:
        - **VCL application**
        - **FireMonkey application**
     3. Keep the default values to use **HTTP** communication protocol and press **Next**.

   Note: Click **Test Port** to check that the port is free.

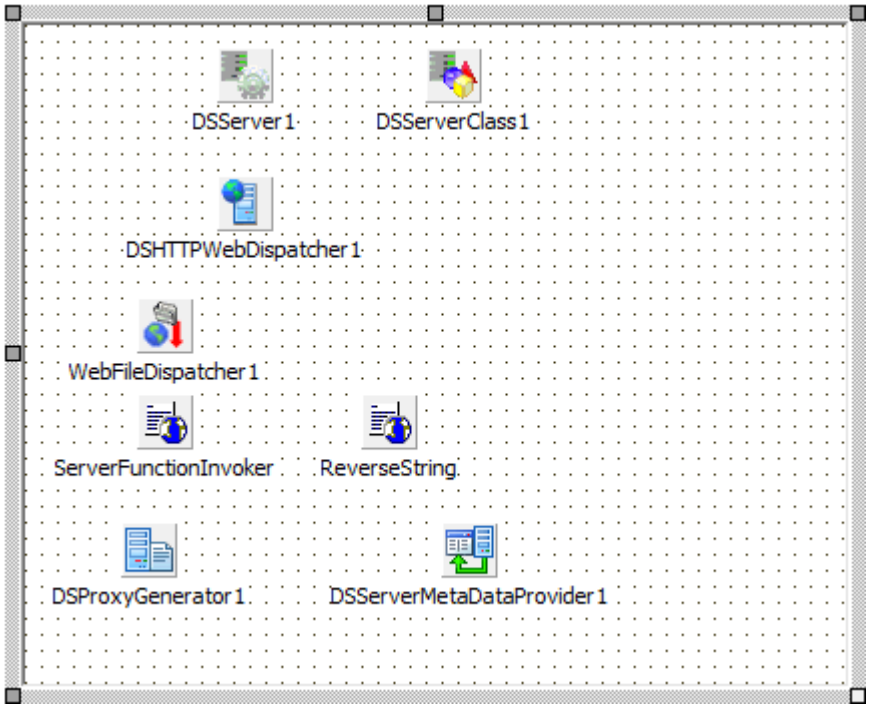
     4. Leave the options selected by default, **Server Methods Class** and the Samples if you want to have an example of methods declaration on the server.
     5. Choose **TDataModule** to entirely implement the server class and click **Next**. This option adds a form where you can place the FireDAC components to connect to the database.
     6. Choose the project location, and the project folder name, and click **Finish**.
2. Change the *Caption* property of the **Form1** to *ServerForm*.

3. Click the main menu item **File > Save All**.

4. Save the **FormUnit1** as *ServerUnit*, the **WebModuleUnit1** as *WebModuleUnit1*, the **ServerMethodsUnit1** as *ServerMethodsUnit1*, and save the project as *MyServerProj*.

Key DataSnap Server Components

When you create the **DataSnap REST Application**, all the DataSnap components needed are automatically added to the **WebModuleUnit1** unit.



The main components on the **WebModuleUnit1** unit are:

- TDSServer
- TDSServerClass

The TDSServer component is the logical heart of the DataSnap server application. It contains the Start and Stop methods for starting and stopping the server. You need only one TDSServer component per server application.

The TDSServerClass component represents a server class. The DataSnap server automatically creates and destroys instances of server classes.

The **HTTP** communication protocol provides the communication between the client and the server.

Adding FireDAC components to the Server Module

Add the following components to the **ServerMethodsUnit1** unit:

- A TFDConnection component. Set its *Name* property to "FDConnectionEMPLOYEE".
  - Right-click the component and click **Connection Editor**. The **FireDac Connection Editor** window opens where you have to define the connection parameters.
  - Select **IB** from the drop-down menu for the **Driver ID** field.
  - Set the path to the Interbase database: C:\Users\Public\Documents\Embarcadero\Studio\15.0\Samples\Data\EMPLOYEE.GDB.
  - Set the User_Name and Password. The values by default for these parameters are User_Name = sysdba and Password = masterkey.
  - Click **OK** to save the changes.
  - On the **Object Inspector** change the **LoginPrompt** property to **False**.
  - Set the **Connected** property to **True**.
- A TFDPhysIBDriverLink component to connect to an InterBase database.
- A TFDGUIxWaitCursor component.
- Three TFDQuery components.
  - Right-click the TFDQuery component and click **Query Editor** to open the **FireDAC Query Editor** window to introduce the SQL statements as detailed later.
  - Rename one component to **FDQueryDepartmentNames** and Introduce the following SQL statement: **select dept_no, department from department** and click OK. This query returns the list of department numbers and names. The result of this query is used at the client to build the list of departments. In order to retrieve this information the client app needs to call the "GetDepartmentNames" server method.
  - Rename the second component to **FDQueryDepartment** and introduce the following SQL statement: **select * from department where DEPT_NO = :DEPT** and click OK.
  - Rename the last component to **FDQueryDepartmentEmployees** and introduce the following statement: **select * from employee where dept_no = :DEPT** and click OK.

  Note:: The last two queries - **FDQueryDepartment** and **FDQueryDepartmentEmployees** - are used to retrieve more detailed information for a given department from DEPARTMENT and from EMPLOYEE tables. The information returned from both queries is exposed to clients via "GetDepartmentEmployees" server method.

- A **TFDStanStorageJSONLink** component.
- A **TFDStanStorageBinLink** component.

The **ServerMethodsUnit1** contains the implementation of two simple methods called **EchoString** and **ReverseString**, which return the value given as a parameter in normal respective reversed states. These are just example methods.

In this example, we add new methods to **ServerMethodsUnit1** to retrieve JSON data from the underlying InterBase **EMPLOYEE** sample database, and the server exposes these methods to the client.

"GetDepartmentNames" server method

- **In Delphi:**

This method gets all departments and give a TFDJSONDataSets as a result.

```
{$METHODINFO ON}
public
    { Public declarations }
function GetDepartmentNames: TFDJSONDataSets;
{$METHODINFO OFF}
```

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

Write code for the function you just added.

```
//GetDepartmentNames
function TServerMethods1.GetDepartmentNames: TFDJSONDataSets;
begin
  // Clear active so that query will reexecute.
  FDQueryDepartmentNames.Active := False;
  Result := TFDJSONDataSets.Create;
  TFDJSONDataSetsWriter.ListAdd(Result,
FDQueryDepartmentNames); // The "TFDJSONDataSetsWriter" class
provides static "ListAdd" method that is using reflection to
convert results of the query into "TFDJSONDataSets".
end;
```

> **Note:** The `{$METHODINFO ON}` directive causes the generation of run-time information needed by the Datasnap server, so this is required--it is not just a comment!
> See METHODINFO directive (Delphi) for more information.

The function returns a new data type: **TFDJSONDataSets**. To use this type you need to include a new unit in the uses section: **Data.FireDACJSONReflect**.

- **In C++:**

This method gets all departments and give a TJSONObject as a result.

```
public:          // User declarations
    TJSONObject* GetDepartmentNames();
```

```
TJSONObject* TServerMethods1::GetDepartmentNames()
{
    FDQueryDepartmentNames->Close();

    TFDJSONDataSets *ds = new TFDJSONDataSets();
    TFDJSONDataSetsWriter::ListAdd(ds,
FDQueryDepartmentNames);

    TJSONObject *obj = new TJSONObject();
    TFDJSONInterceptor::DataSetsToJSONObject(ds, obj);
    return obj;
}
```

You need to include these units:

1. include "System.Json.hpp" //TJSONObject
2. include "Data.FireDACJSONReflect.hpp" //TFDJSONDataSets

"GetDepartmentEmployees" server method

Use this method to retrieve more detailed information for a given department from DEPARTMENT and from EMPLOYEE tables using FDQueryDepartment and FDQueryDepartmentEmployees. We can get the data from both queries using just one method. For this purpose we need to create two constants.

- **In Delphi:**

```
//Include the constants under the uses section of the
implementation.
const
  sDepartment = 'Department';
  sEmployees = 'Employees';
```

```
{$METHODINFO ON}
public
    { Public declarations }
function GetDepartmentEmployees(const AID: string):
TFDJSONDataSets;
```

```
{$METHODINFO OFF}
```

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

Write code for the function you just added.

```
function TServerMethods1.GetDepartmentEmployees(const AID:
string): TFDJSONDataSets;
begin
  // Clear active so that query will reexecute.
  FDQueryDepartmentEmployees.Active := False;
  FDQueryDepartment.Active := False;
  FDQueryDepartment.Params[0].Value := AID;
  FDQueryDepartmentEmployees.Params[0].Value := AID;

   // Create dataset list
  Result := TFDJSONDataSets.Create;
   // Add departments dataset
  TFDJSONDataSetsWriter.ListAdd(Result, sDepartment,
FDQueryDepartment);
    // Add employees dataset
  TFDJSONDataSetsWriter.ListAdd(Result, sEmployees,
FDQueryDepartmentEmployees);
end;
```

These two methods are used by a client app to receive information about departments. First we get the list of department names and their IDs (GetDepartmentNames method). When a client selects a department from the list, then the detailed information about department and its employees is returned from the second server method (GetDepartmentEmployees).

- **In C++:**

```
const System::String sEmployees = "Employees";
const System::String sDepartment = "Department";


public:          // User declarations
    TJSONObject* GetDepartmentEmployees(System::UnicodeString
AID);


// Get a Department and all Employees in the department.
Return TJSONObject.
```

```
TJSONObject*
TServerMethods1::GetDepartmentEmployees(System::UnicodeString
AID)
{
  FDQueryDepartmentEmployees->Active = false;
  FDQueryDepartment->Active = false;
  FDQueryDepartment->Params->operator [](0)->Value = AID;
  FDQueryDepartmentEmployees->Params->operator [](0)->Value =
AID;

  // Create dataset list
  TFDJSONDataSets *ds = new TFDJSONDataSets();
  // Add departments dataset
  TFDJSONDataSetsWriter::ListAdd(ds, sDepartment,
FDQueryDepartment);
  // Add employees dataset
  TFDJSONDataSetsWriter::ListAdd(ds, sEmployees,
FDQueryDepartmentEmployees);

  TJSONObject *obj = new TJSONObject();
  TFDJSONInterceptor::DataSetsToJSONObject(ds, obj);
  return obj;
}
```

"ApplyChangesDepartmentEmployees" server method

Use this method to send data updates from client and updating the underlying database. This method does not return any value. This method uses a TFDJSONDeltas parameter. This type is also included in the **Data.FireDACJSONReflect** unit. In just one operation we can update multiple tables. Here is the source code.

- **In Delphi:**

```
{$METHODINFO ON}
public
    { Public declarations }
    procedure ApplyChangesDepartmentEmployees(const
ADeltaList: TFDJSONDeltas);
{$METHODINFO OFF}
```

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

Write code for the function you just added.

```
// Update department and employees using deltas
procedure TServerMethods1.ApplyChangesDepartmentEmployees(
  const ADeltaList: TFDJSONDeltas);
var
  LApply: IFDJSONDeltasApplyUpdates;
begin
  // Create the apply object
  LApply := TFDJSONDeltasApplyUpdates.Create(ADeltaList);
  // Apply the department delta
  LApply.ApplyUpdates(sDepartment, FDQueryDepartment.Command);
  if LApply.Errors.Count = 0 then
    // If no errors, apply the employee delta
    LApply.ApplyUpdates(sEmployees,
FDQueryDepartmentEmployees.Command);
  if LApply.Errors.Count > 0 then
    // Raise an exception if any errors.
    raise Exception.Create(LApply.Errors.Strings.Text);
end;
```

- **In C++**

```
public:          // User declarations
    void ApplyChangesDepartmentEmployees(TJSONObject*
AJSONObject);


void
TServerMethods1::ApplyChangesDepartmentEmployees(TJSONObject*
AJSONObject)
{
  TFDJSONDeltas *LDeltas = new TFDJSONDeltas();
  TFDJSONInterceptor::JSONObjectToDataSets(AJSONObject,
LDeltas);

  TFDJSONErrors *errs = new TFDJSONErrors();

  // Apply the department delta
  TFDJSONDeltasApplyUpdates::ListApplyUpdates(LDeltas,
sDepartment, FDQueryDepartment->Command, errs);

  // If no errors, apply the employee delta
  if (errs->Count == 0) {
    TFDJSONDeltasApplyUpdates::ListApplyUpdates(LDeltas,
sEmployees, FDQueryDepartmentEmployees->Command, errs);
  }
```

```
  // Raise an exception if any errors.
  if (errs->Count > 0) {
     throw new Exception(errs->Strings->Text);
  }

}
```

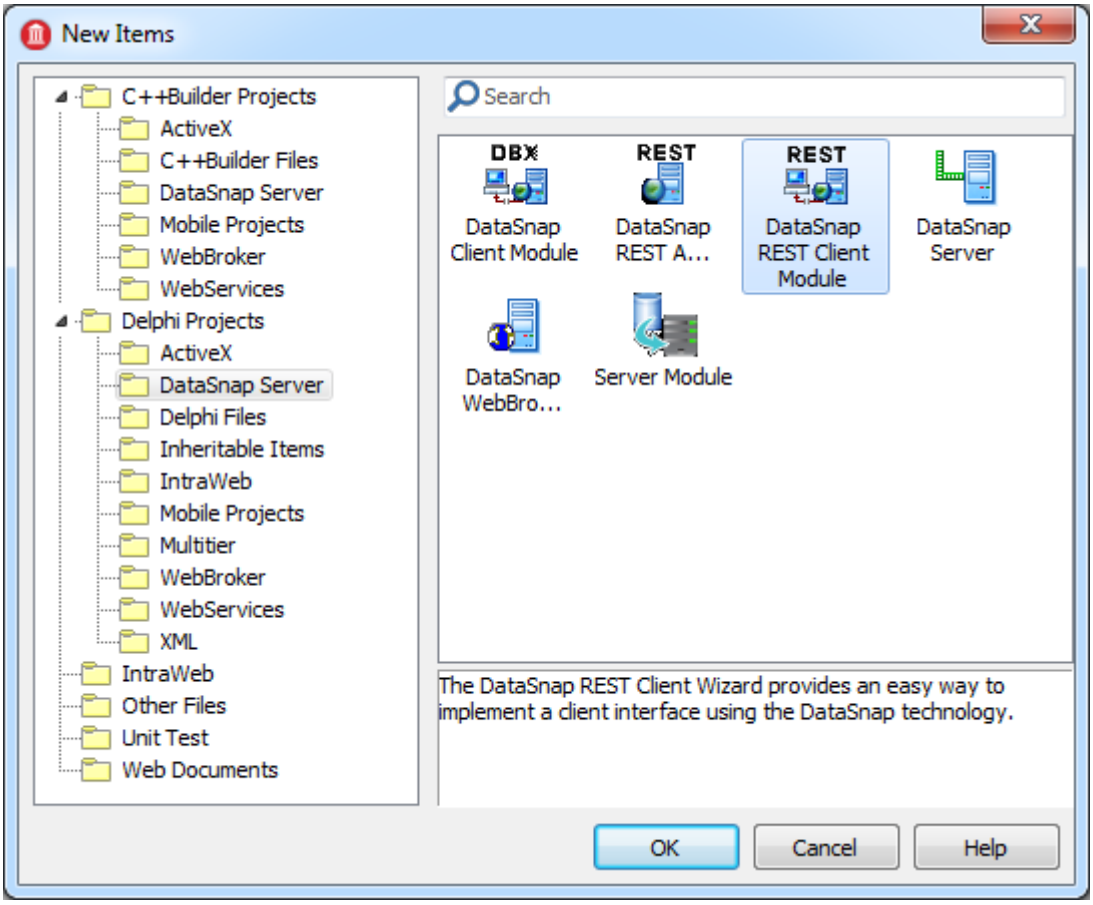
Before creating the Client application, run the server:

- Choose **Run > Run Without Debugging** from the main menu.
- Click the **Start** button. You can minimize the **ServerForm** dialog that displays.

This step is very important because you need a connection to the server to create the client classes needed to retrieve the data from the server.

Creating the Client Application

1. To create the client application in the same project group as the server application, follow the steps below:
   - In the **Project Manager**, right-click the **ProjectGroup**.
   - Select the **Add New Project** option.
   - From the Delphi or the C++Builder Projects item, select **FireMonkey Desktop Application**, and click **OK**.
   - Choose **HD Desktop Application** and click **OK** .
   - Click the main menu item **File > New > Other**.
   - From the **DataSnap Server** node in the left column, select **DataSnap REST Client Module** and press **OK**.



- Specify the type of the module:
  - Keep the default value selected--**Local server**--and press **Next**.
  - Keep the default value selected--**DataSnap stand alone server**--and press **Next**.
  - Keep the default value for the connection parameters and press **Finish**.
    - **Note:** Click **Test Connection** to check the connection with the server. If the server is not running you are not able to finish the wizard.

Now the wizard generates *ClientModuleUnit* and *ClientClassesUnit*.

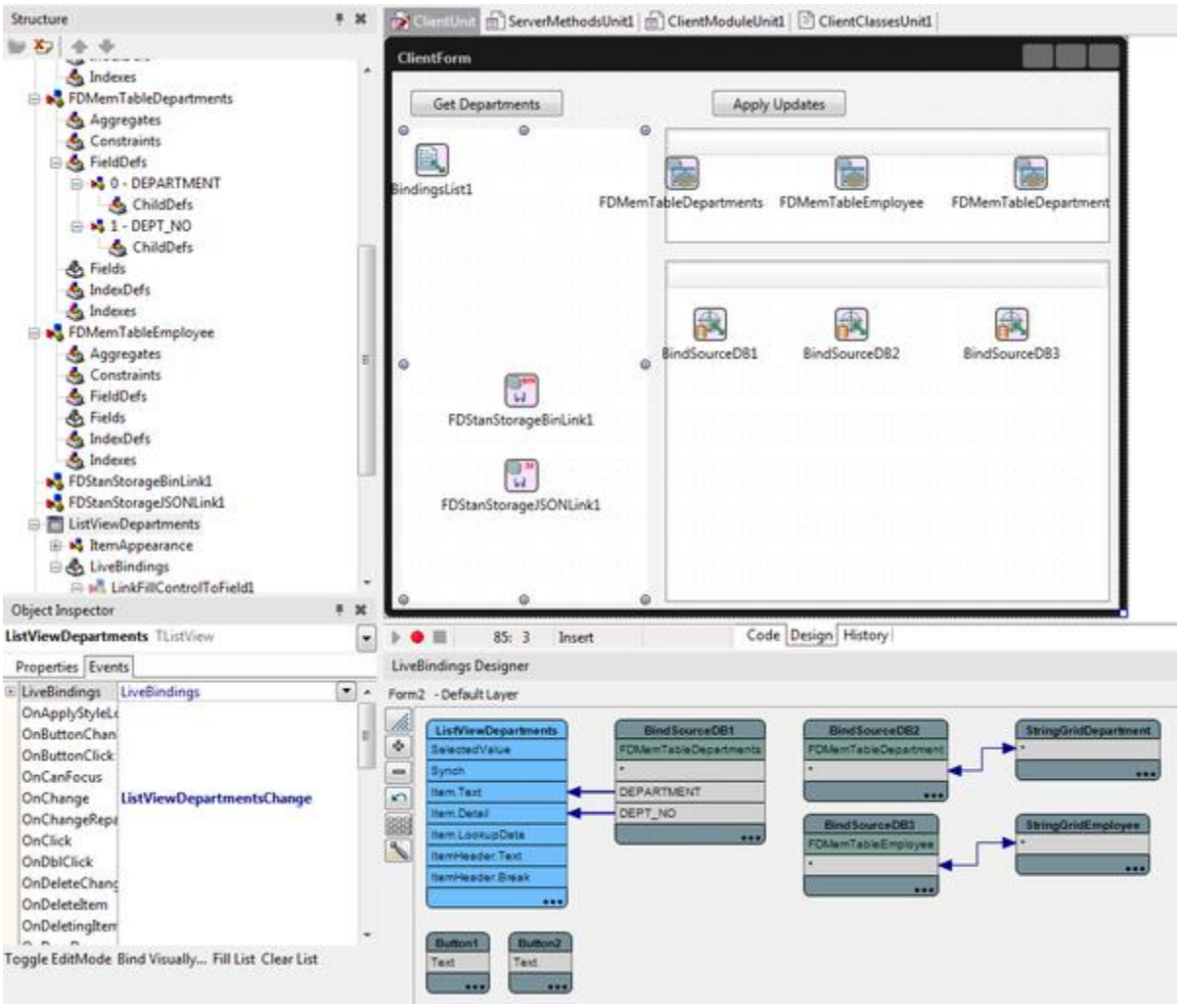
2. Change the *Caption* property of the **Form2** to *ClientForm*.
3. Click the main menu item **File > Save All**.
4. Save the **Unit2** as *ClientUnit*, the **ClientModuleUnit1** as *ClientModuleUnit1*, the **ClientClassesUnit1** as *ClientClassesUnit1*, and the project as *MyClientProj*.
5. Save the Project Group to *DSServerExample*.
6. Populate the client form with the following controls and components:
   - A TButton component. Change the **text** property to Get Departments.
   - A TButton component. Change the **text** property to Apply Updates.

- A TListView component. Change the **name** property to ListViewDepartments.
- A TStringGrid component. Change the **name** property to StringGridDepartment.
- A TStringGrid component. Change the **name** property to StringGridEmployee.
- A TFDMemTable component. Change the **name** property to FDMemTableDepartments. This dataset is used only for storing the read-only data with the list of department numbers and their names.
  - You need field definitions in the TFDMemTable to bind them at design time. On the Structure panel, right-click **FieldDefs** and click **Add Item** to add two new items: 0 - DEPARTMENT, and 1 - DEPT_NO.
    - **Note:** Field types are not important, but the names of the field definitions need to correspond to the fields in datasets received from the server.
- A TFDMemTable component. Change the **name** property to FDMemTableDepartment.
  - Change the **CachedUpdates** property to **True**.
- A TFDMemTable component. Change the **name** property to FDMemTableEmployee.
  - Change the **CachedUpdates** property to **True**.
- A TFDStanStorageJSONLink component
- A TFDStanStorageBINLink component.

Using the LiveBindings Designer

Use visual live bindings to bind data to visual controls.

- Go to **View > LiveBindings Designer** to open the **LiveBindings Designer** window.
- Change the **ItemAppearance > ItemAppearance** property of the TListView to **ListItemRightDetail** to add the **Item.Detail** property to the TListView.
- Bind the DEPARTMENT field of the FDMemTableDepartments to **Item.Text**.
- Bind the DEPT_NO field of the FDMemTableDepartments to **Item.Detail**.
- Bind the StringGridDepartment with FDMemTableDepartment.
- Bind the StringGridEmployee with FDMemTableEmployee.



Calling the Server Methods on the Client

You need to call the server methods on the client application to retrieve the data.

Note: If you change the server methods definition, you need to restart the server. After restarting the server, you also need to refresh the proxy, right-click the TDSRestConnection component from ClientModuleUnit and click **Generate DataSnap client classes** in the context menu.

GetDepartmentNames

In Delphi:

In the ClientUnit Form include ClientModuleUnit1. Add it to the uses section.

```
private
    { Private declarations }
    procedure GetDepartmentNames;
```

```
procedure TForm2.GetDepartmentNames;
var
    LDataSetList: TFDJSONDataSets; //To use this type
you need to include a new unit in the uses section:
Data.FireDACJSONReflect.
begin
    FDMemTableDepartments.Close; // It empties the
memory table of any existing data before adding the new
context.
    // Get dataset list containing Employee names
    LDataSetList :=
ClientModule1.ServerMethods1Client.GetDepartmentNames;
    // Reads the first and only dataset, number 0.
    FDMemTableDepartments.AppendData(
      TFDJSONDataSetsReader.GetListValue(LDataSetList,
0));  //It uses a reader from The
"TFDJSONDataSetsWriter" class to populate the memory
table from the dataset list class.

    FDMemTableDepartments.Open;
end;
```

Implement the *OnClick* event to call the GetDepartmentNames procedure.

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  GetDepartmentNames; //Calling this method populates
the TListView.
end;
```

- **In C++:**

Include the following units to your project:

1. include <Memory> // This header defines general utilities to manage dynamic memory.
2. include "Data.FireDACJSONReflect.hpp"
3. include "DataSnap.DSClientREST.hpp"
4. include "ClientModuleUnit1.h"

```
public:           // User declarations
    void GetDepartmentNames();

void TForm2::GetDepartmentNames() {
```

```
     TJSONObject* LJSONObject (ClientModule1-
>ServerMethods1Client->GetDepartmentNames()); // Gets
JSON data from the server using "ClientModule1" class
that was generated with the "DataSnap REST Client
Module" wizard.
     std::auto_ptr<TFDJSONDataSets>LDataSets(new
TFDJSONDataSets()); //Automatic Pointer.
     TFDJSONInterceptor::JSONObjectToDataSets(LJSONObject
, LDataSets.get()); //Converts JSON to a dataset, just
the opposite that in the server.
     FDMemTableDepartments->Active = false;
     TFDAdaptedDataSet * LDataSet =
TFDJSONDataSetsReader::GetListValue (LDataSets.get(),
0);
     FDMemTableDepartments->AppendData(*LDataSet);
//Appends the DataSet to FDMemTableDepartment table.
}

void __fastcall TForm2::Button1Click(TObject *Sender)
{
     GetDepartmentNames(); //Calling this method
populates the TListView.
}
```

GetDepartmentEmployees

This method is used to download department and employee detailed information for a selected department. The code is executed when clicking on the department names from the TListView.

```
const
  sEmployees = 'Employees';
  sDepartment = 'Department';

private
    { Private declarations }
    procedure GetDepartmentEmployees(const ADEPTNO:
string);

procedure TForm2.GetDepartmentEmployees(const ADEPTNO:
string);
var
    LDataSetList: TFDJSONDataSets;
    LDataSet: TFDDataSet;
begin
```

```
    LDataSetList :=
ClientModule1.ServerMethods1Client.GetDepartmentEmployee
s(ADEPTNO);
    // Get department dataset
    LDataSet :=
TFDJSONDataSetsReader.GetListValueByName(LDataSetList,sD
epartment);
    // Update UI
    FDMemTableDepartment.Active := False;
    FDMemTableDepartment.AppendData(LDataSet);

    // Get employees dataset
    LDataSet :=
TFDJSONDataSetsReader.GetListValueByName(LDataSetList,
sEmployees);
    // Update UI
    FDMemTableEmployee.Active  := False;
    FDMemTableEmployee.AppendData(LDataSet);
end;
```

Implement the *OnChange* event from the TListView to call the GetDepartmentEmployees procedure.

```
procedure TForm2.ListViewDepartmentsChange(Sender:
TObject);
var
  LDEPTNO: string;
begin
  // Show department/employee details
  LDEPTNO := ListViewDepartments.Selected.Detail;
  GetDepartmentEmployees(LDEPTNO);
end;
```

- **In C++:**

```
const System::String sEmployees = "Employees";
const System::String sDepartment = "Department";

public:           // User declarations
    void GetDepartmentEmployees(const System::String
ADEPTNO);

void TForm2::GetDepartmentEmployees(const System::String
ADEPTNO)
{
```

```
            TJSONObject* LJSONObject(ClientModule1-
>ServerMethods1Client->GetDepartmentEmployees
(ADEPTNO));
            std::auto_ptr<TFDJSONDataSets> LDataSets(new
TFDJSONDataSets());

     TFDJSONInterceptor::JSONObjectToDataSets(LJSONObject
, LDataSets.get());

            { //multiple declaration for 'LDataSet'
            TFDAdaptedDataSet * LDataSet =
TFDJSONDataSetsReader::GetListValueByName(LDataSets.get(
), sDepartment);
            // Update UI
            FDMemTableDepartment->Active = False;
            FDMemTableDepartment->AppendData(*LDataSet);
            }

            { //multiple declaration for 'LDataSet'
            TFDAdaptedDataSet * LDataSet =
TFDJSONDataSetsReader::GetListValueByName(LDataSets.get(
), sEmployees);
            // Update UI
            FDMemTableEmployee->Active = False;
            FDMemTableEmployee->AppendData(*LDataSet);
        }
}
```

Implement the *OnChange* event from the TListView to call the GetDepartmentEmployees procedure.

```
void __fastcall
TForm2::ListViewDepartmentsChange(TObject *Sender)
{
    // Show department/employee details
     System::String LDEPTNO = ListViewDepartments-
>Selected->Detail;
     GetDepartmentEmployees(LDEPTNO);

}
```

Procedure ApplyUpdates

The data stored in the "FDMemTableDepartment" and "FDMemTableEmployee" can be modified through the user interface. When the end user clicks on the "Apply Updates" button the following code is used to send the updates back to the server.

- **In Delphi:**

First you need to create a function to get the TFDJSONDeltas:

```
private
    { Private declarations }
    function GetDeltas: TFDJSONDeltas;

function TForm2.GetDeltas: TFDJSONDeltas;
begin
  // Post if editing
  if FDMemTableDepartment.State in dsEditModes then
  begin
    FDMemTableDepartment.Post;
  end;

  if FDMemTableEmployee.State in dsEditModes then
  begin
    FDMemTableEmployee.Post;
  end;

  // Create a delta list
  Result := TFDJSONDeltas.Create;
  // Add deltas
  TFDJSONDeltasWriter.ListAdd(Result, sEmployees,
FDMemTableEmployee);
  TFDJSONDeltasWriter.ListAdd(Result, sDepartment,
FDMemTableDepartment);
end;
```

Now create the method to apply the changes:

```
private
    { Private declarations }
    procedure ApplyUpdates;

procedure TForm2.ApplyUpdates;
var
  LDeltaList: TFDJSONDeltas;
begin
  LDeltaList := GetDeltas;
```

```
  // Call server method.  Pass the delta list.

ClientModule1.ServerMethods1Client.ApplyChangesDepartmen
tEmployees(LDeltaList);

end;
```

Implement the *OnClick* event to call the ApplyUpdates procedure.

```
procedure TForm2.ButtonApplyUpdatesClick(Sender:
TObject);
begin
  ApplyUpdates;
end;
```

- **In C++:**

```
public:           // User declarations
    void ApplyUpdates();

void TForm2::ApplyUpdates()
{
     // Post if editing
     if (dsEditModes.Contains(FDMemTableDepartment-
>State))
     {
            FDMemTableDepartment->Post();
     }

     if (dsEditModes.Contains(FDMemTableEmployee->State))
     {
            FDMemTableEmployee->Post();
     }

     // Create a delta list
     TFDJSONDeltas * LDeltas = new TFDJSONDeltas();
     // Add deltas
     TFDJSONDeltasWriter::ListAdd(LDeltas, sEmployees,
FDMemTableEmployee);
     TFDJSONDeltasWriter::ListAdd(LDeltas, sDepartment,
FDMemTableDepartment);

     TJSONObject * LJSONObject(new TJSONObject());
     TFDJSONInterceptor::DataSetsToJSONObject(LDeltas,
LJSONObject);
```

```
    // Call server method.  Pass the delta list.
    ClientModule1->ServerMethods1Client-
>ApplyChangesDepartmentEmployees(LJSONObject);

}
```

Implement the *OnClick* event to call the ApplyUpdates procedure.

```
void __fastcall TForm2::Button2Click(TObject *Sender)
{
    ApplyUpdates();
}
```