# LANSA OPEN FOR .NET 3.8

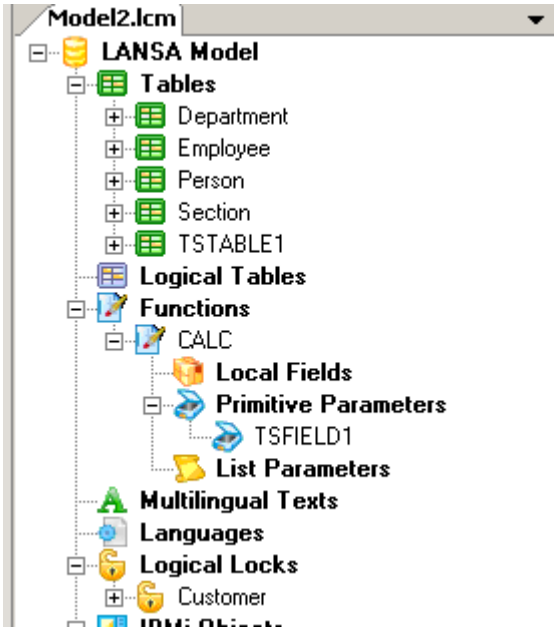
## QUICK START GUIDE

## What is LANSA Open for .NET 3.8?

LANSA Open for .NET 3.8 is a LANSA Development Environment for .NET that is integrated with Microsoft Visual Studio.

- It allows .NET applications to easily make use of the objects in a LANSA Repository (tables, functions, validation rules, multilingual texts) in an object-based manner.
- It does **not** however allow the creation and modification of objects in a LANSA Repository.

.NET developers typically create **a LANSA data model** which specifies which LANSA objects they want to make accessible from their .NET code. The LANSA data model file will be part of their C# or VB.NET projects and behind the scenes LANSA Open for .NET (integrated into Visual Studio) will automatically create the classes and methods that represent the objects in this model, which are immediately available for use from any .NET code.




LANSA Open for .NET 3.8 consists of 2 main components:

- **LANSA Repository Explorer**

  The Repository Explorer allows .NET developers to connect to a LANSA repository on a remote server and inspect the objects defined in the repository and view/edit content of tables.
- **LANSA Data Model Editor**

  The Data Model Editor allows .NET developers to visually construct a LANSA data model (LANSA tables, server functions, etc) for their application. This is done by dragging LANSA objects from the Repository Explorer and dropping them on the editor. When saved this visual representation of the data model will be transformed automatically into .NET classes and methods (behind the scenes) and will be readily available for use by the .NET developers. This in conjunction with Visual Studio IntelliSense makes programming with LANSA objects much simpler and less prone to mistakes.

## Why LANSA Open for .NET 3.8?

- Eliminates the need for .NET developers to use Visual LANSA, something they are not familiar with and will require a considerable amount of time to get used to.
- It provides a programming model that .NET developers are familiar with.
- Effective collaboration between LANSA developers and .NET developers:
    - LANSA developers create a data model using standalone LANSA data model editor.
    - The data model will be passed to the .NET developers, which instantly allows the .NET developers to easily access the defined LANSA objects from their .NET code.

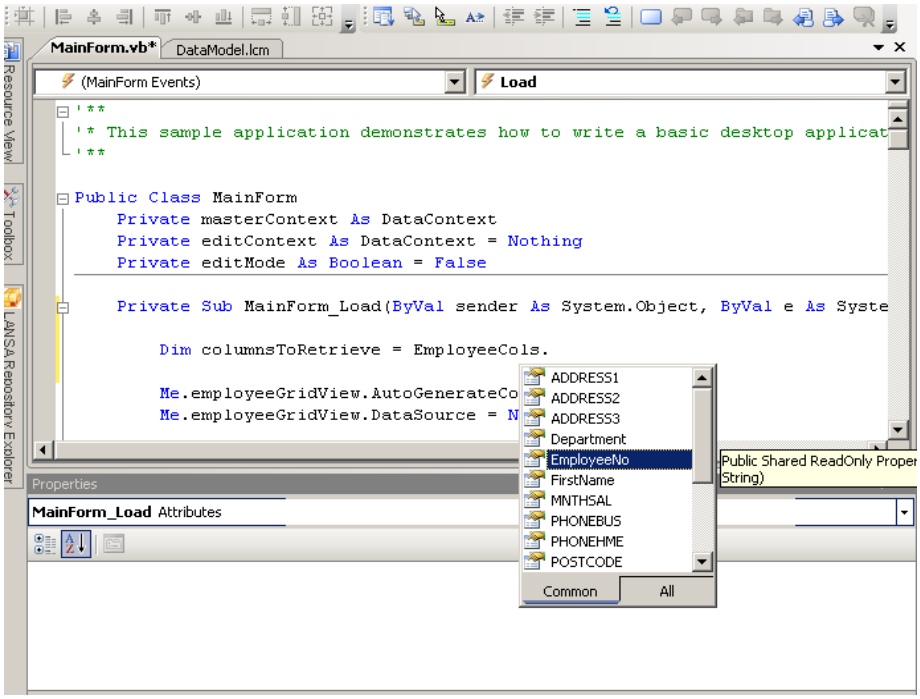
## What Will a Simple C# Code to Retrieve and Update an Employee Record Look Like?

To give you a feeling of what you can expect from LANSA Open for .NET 3.8, here is a very simple example of the usage of the data model after it is designed and saved. You don't have to fully understand the code and what it does, this is just to give you a taste of coding with LANSA Open for .NET 3.8. The classes and properties such as ***Employee***, ***FirstName***, ***EmployeeCols*** are automatically generated for you behind the scenes from the data model.

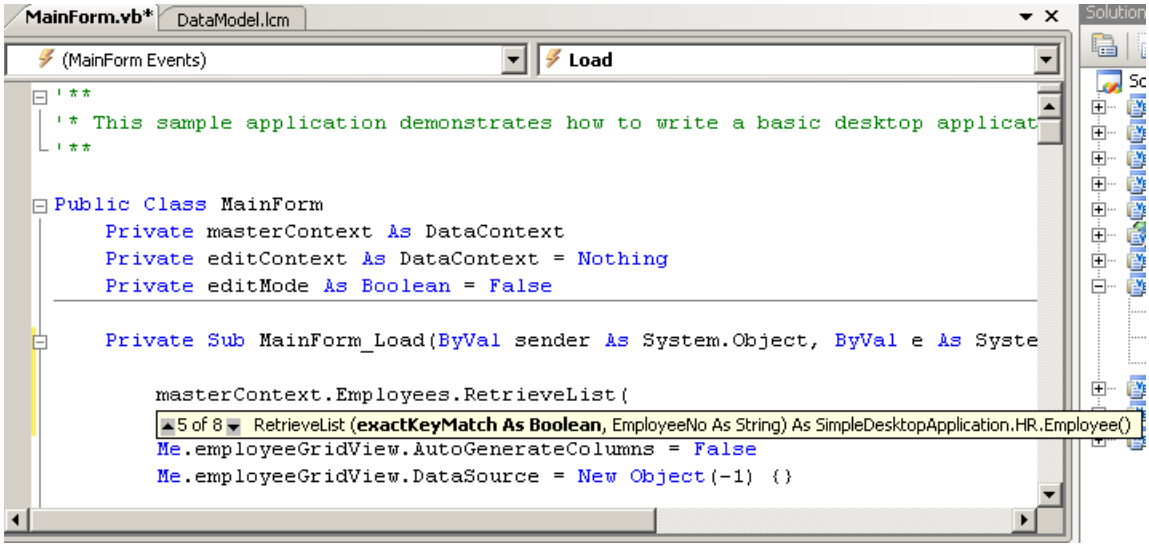
```
// Create a LANSA data context - this is always the starting point
DataContext context = new DataContext(true);

// Retrieving all fields of an employee with key field EMPNO = "A0070"
Employee employee = context.Employees.RetrieveItem("A0070");
Console.WriteLine(employee.FirstName);
Console.WriteLine(employee.Surname);

// Updating the name & salary of the employee
employee.FirstName = "Andrew";
employee.Salary += 10000;

// Commit the changes to the LANSA database
context.SubmitChanges();

// Retrieving all employees with salary bigger than 50,000
Employee[] list1 = context.Employees.RetrieveList(EmployeeCols.Salary > 50000);

// Retrieving only Employee Number and first name fields of all employees
Employee[] list2 = context.Employees.RetrieveList(EmployeeCols.EmpNo +
EmployeeCols.FirstName);
```

## How is Coding Made Easier with LANSA Open for .NET 3.8?

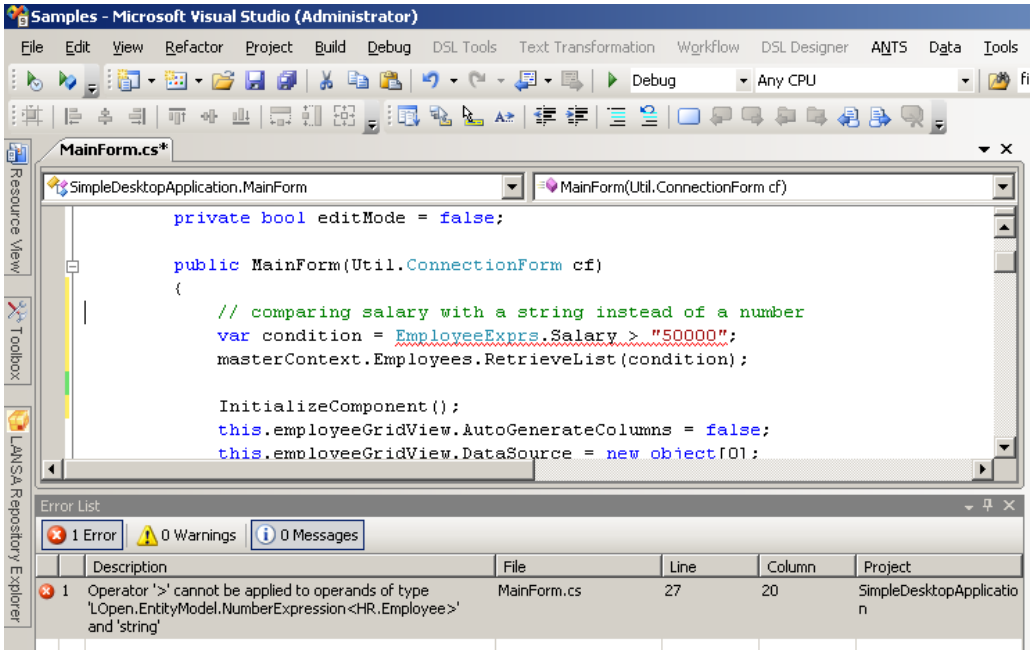
- IntelliSense lets the programmers know what fields are available in a table & what operations can be performed.




- IntelliSense can quickly show the programmers what they can and can't do.
  In this screenshot below, it shows that EmployeeNo is one of the keys in the table and we can find an employee with a certain EmployeeNo.




- When you are trying to do something that is invalid, the compiler will alert you so that you don't have to wait until you run your program to find out that you've done something wrong.




## What Doesn't LANSA Open for .NET 3.8 Do?

It does not deal with UI / form design. So you can't just drag a field from the repository onto a form and expect a default visualization of the field to be created automatically for you as in Visual LANSA.

## Runtime Requirements

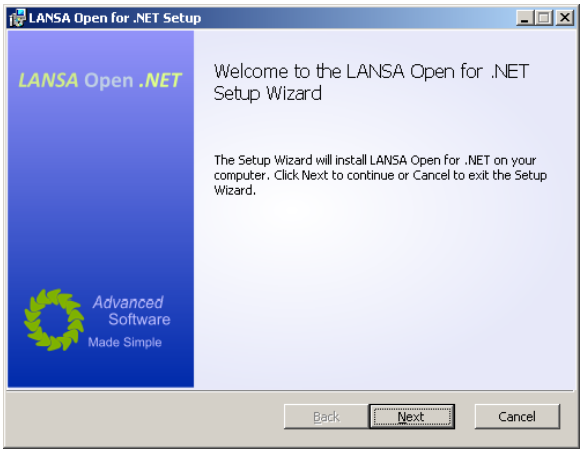
.NET Framework 4.0

## Supported Visual Studio Versions

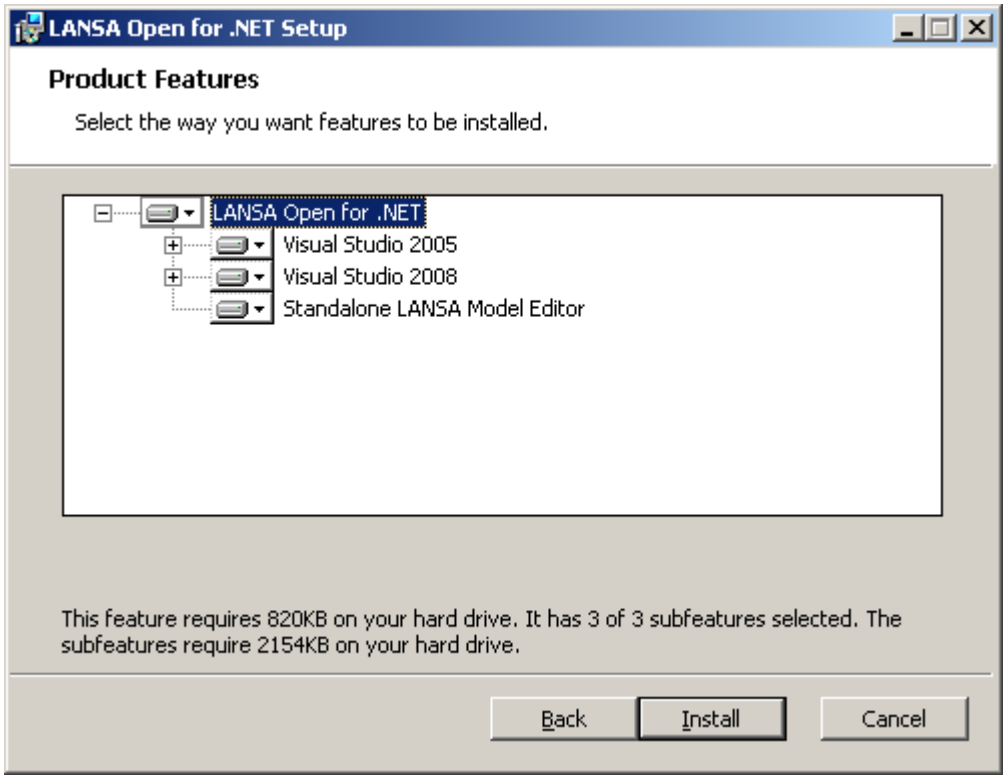
- Visual Studio 2010
- Visual Studio 2012
- Visual Studio 2013
- Visual Studio 2015

## Installing LANSA Open for .NET

Run the LansaNETInstaller.msi




## Components of LANSA Open for .NET




There are two versions of the two tools mentioned earlier (LANSA Repository Explorer and Data Model Editor):

1. Integrated with Visual Studio
2. Standalone application

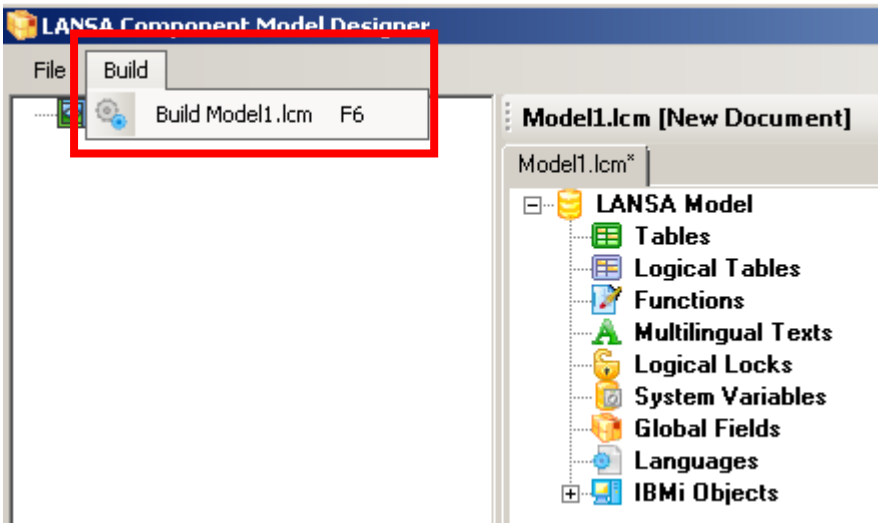
The two versions are exactly the same in terms of functionality (there are some minor differences related to how the data model can be used in the code which will be discussed in the next section). The next section will discuss the circumstances in which the standalone version is necessary or preferable to the Visual Studio integrated version.

## Is Visual Studio Express Supported?

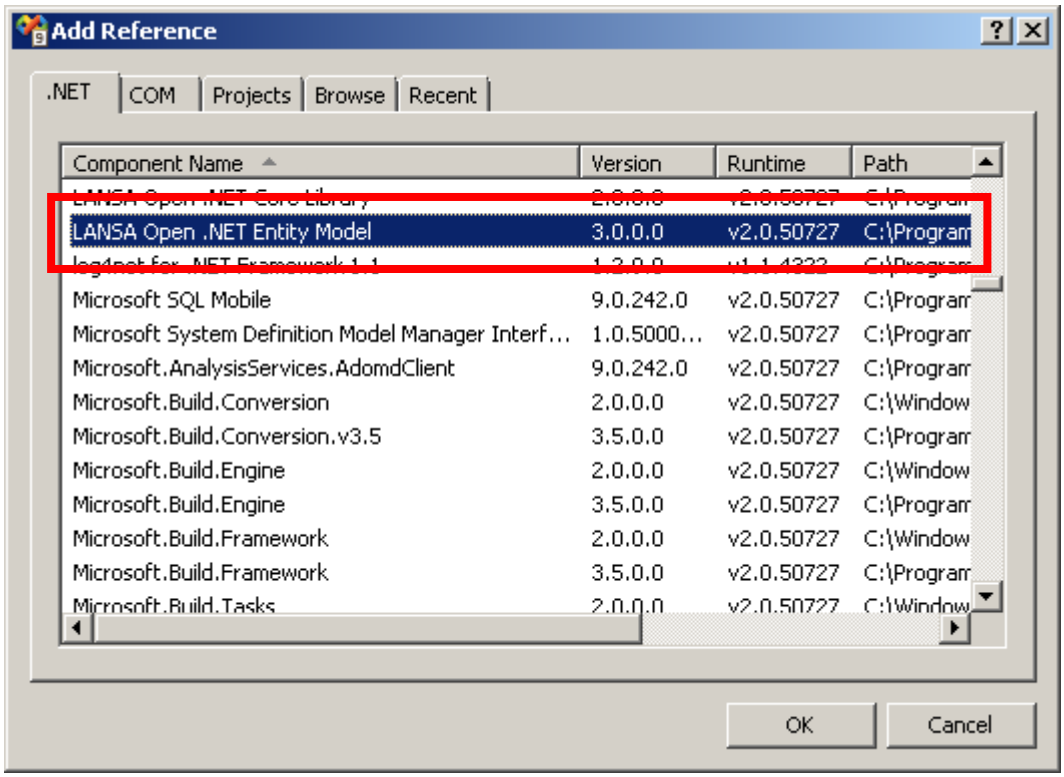
No, however it does not mean that you cannot develop with LANSA Open for .NET 3.8 with Visual Studio Express.

What you need to do when developing with Visual Studio Express:

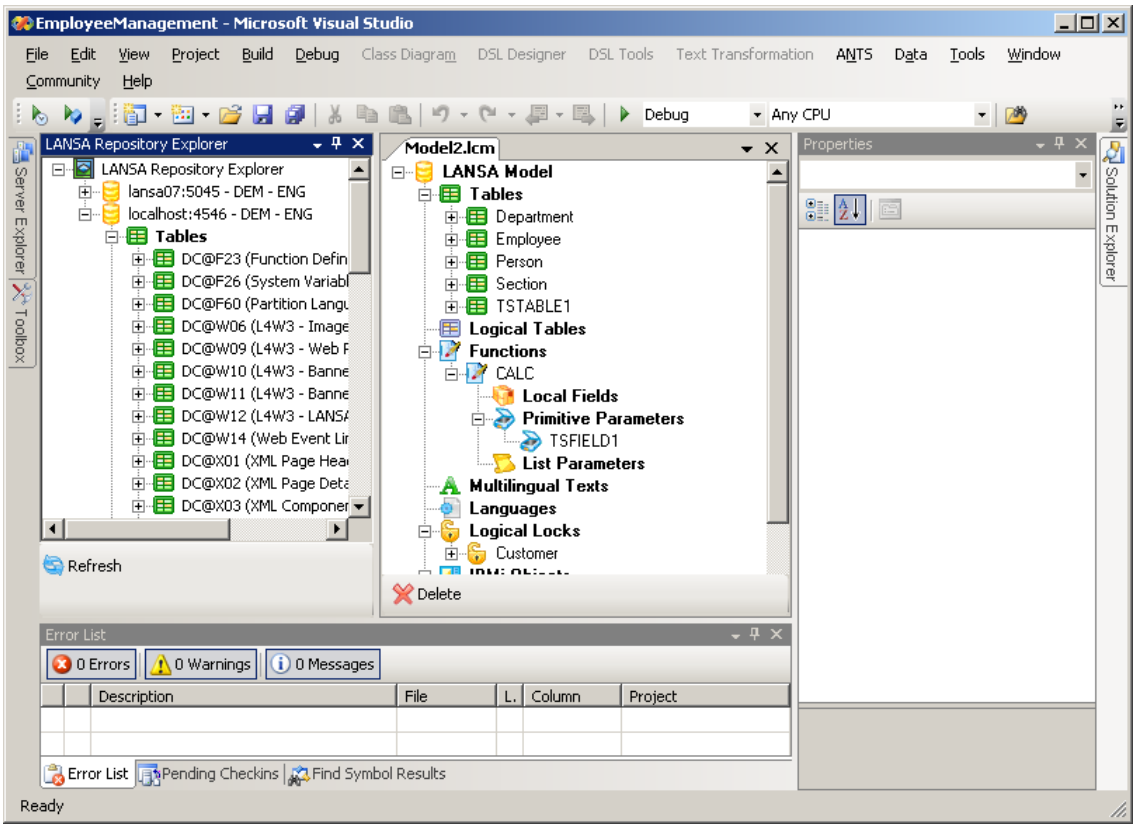
1. Use the Standalone Data Model Editor to create and edit the data model.
2. Build the data model into an assembly (DLL).



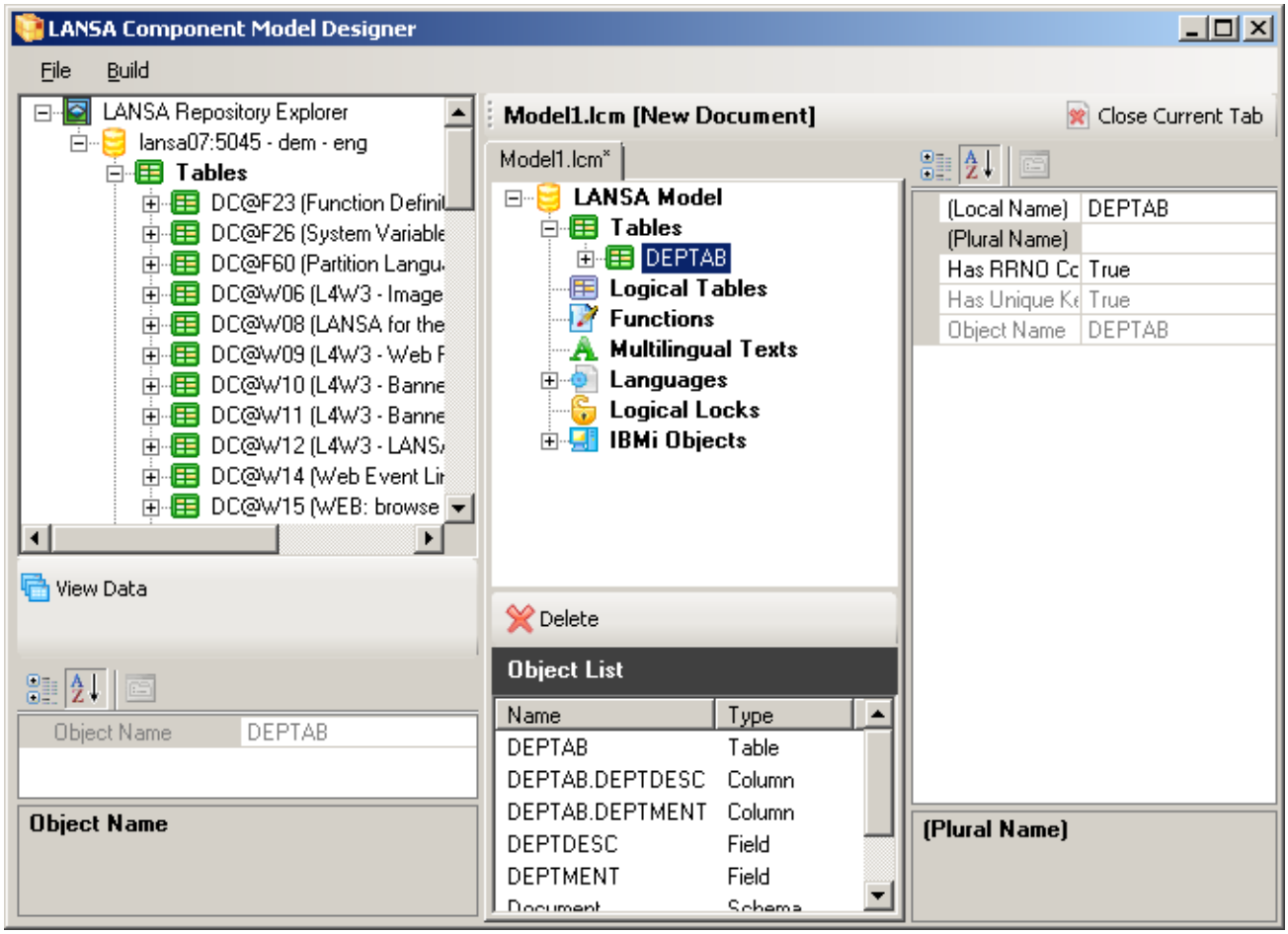

3. From Visual Studio Express, add the following assembly references to your project:
   - The data model assembly created in step 2.
   - **LANSA Open .NET Entity Model** assembly.



## A Glimpse of LANSA Repository Explorer and Data Model Editor




(Visual Studio Integrated version - left is the **Repository Explorer**, middle pane is **Data Model Editor**)




(Standalone version of **LANSA Repository Explorer** and **Data Model Editor**)

## Why Do We Have the Standalone Version of the Data Model Editor?

The standalone version will come in handy when the person or group designing the data model is different from the .NET developers who will be developing the actual application.

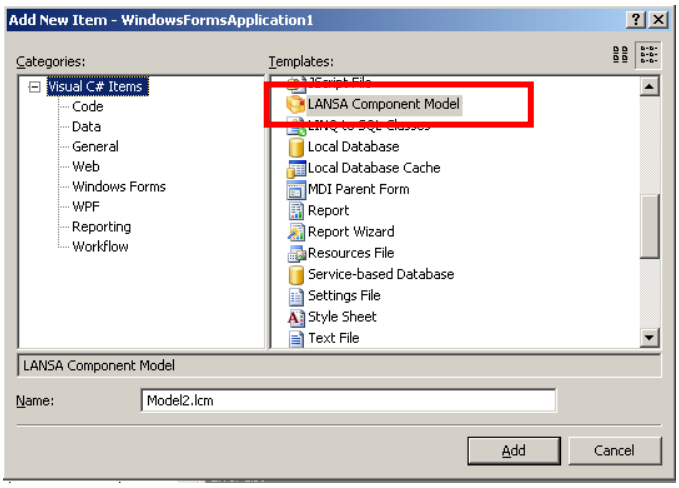
The rationale behind this:

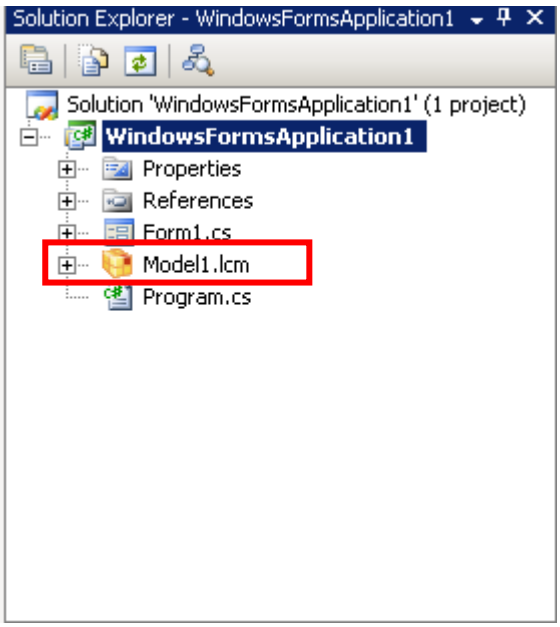
1. For somebody to be able to construct a model of tables, server functions, and other LANSA objects, they need to have good understanding of the objects.
2. This person will most likely be a LANSA developer, but **not** a .NET developer.
3. Since this person is a LANSA developer, most likely he or she will not have a Visual Studio installed in his or her computer.
4. The standalone Data Model Editor enables the LANSA developer to create a LANSA data model, save it as a data model file (.lcm), and then pass the file to a .NET developer who will then incorporate the data model file into their .NET project. Once incorporated into the project, the .NET developers will be able write C# or VB code that makes use the LANSA objects defined in the data model file.

The standalone Data Model Editor is also useful for somebody who does not have Visual Studio since the standalone Data Model Editor is capable of generating a .NET assembly (DLL) from the data model file. This assembly can be referenced from any other .NET applications thus providing access to LANSA objects defined in the data model file.

## What Does the Visual Studio Integrated Version Do That the Standalone Version Does Not?

In the Visual Studio integrated version, it is possible to incorporate the data model file (.lcm) directly into your .NET project (C# or VB.NET project) and behind the scenes the .NET classes and methods that correspond to the objects in the model will be generated automatically every time the model file is saved.

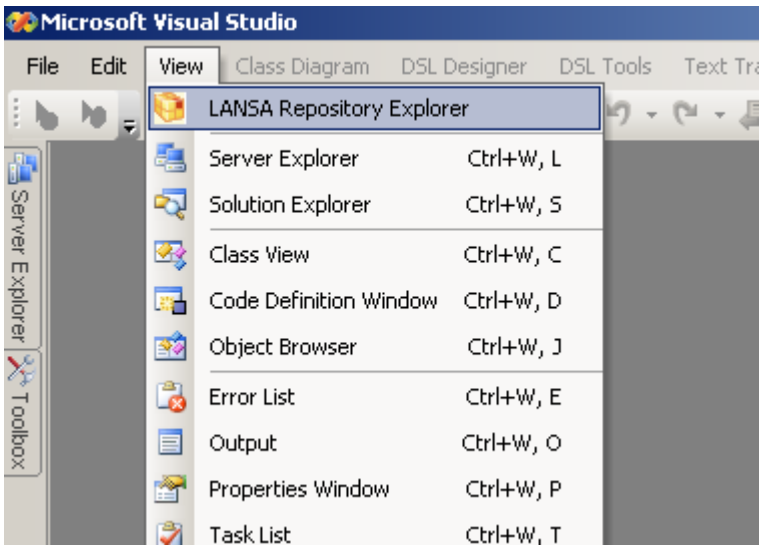


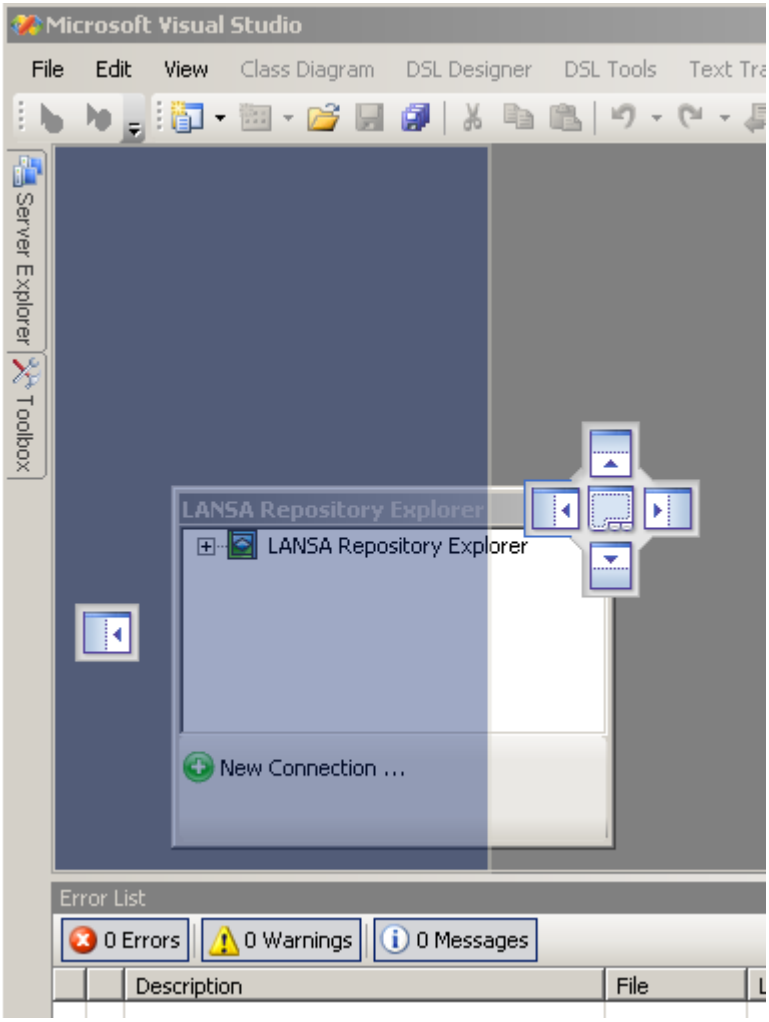

If you are using the standalone data model editor, before it can be used the editor will need to build the data model file to produce a .NET assembly (DLL) that you need to reference from your own code.

## How Do I Start the LANSA Repository Explorer in Visual Studio?

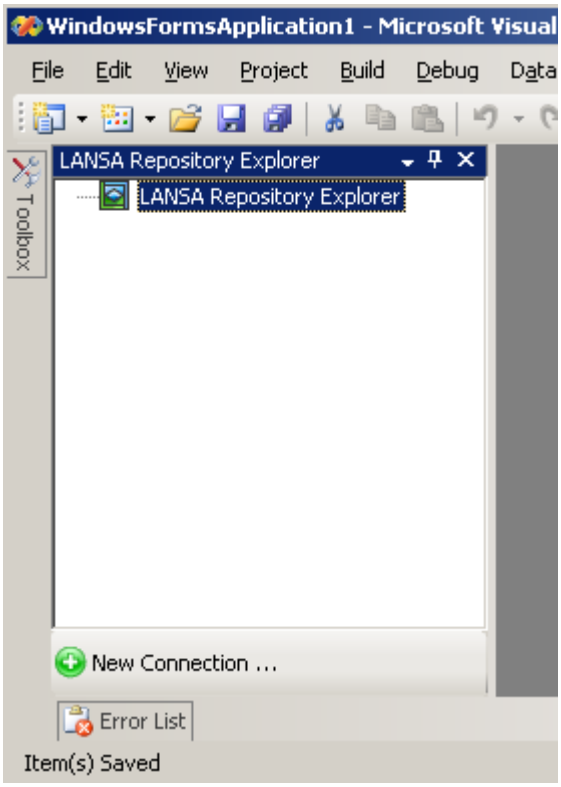
After installing LANSA Open for .NET, under the **View** menu in Visual Studio there will be a new menu item **LANSA Repository Explorer**. Click on that item to open up the **LANSA Repository Explorer** tool window.



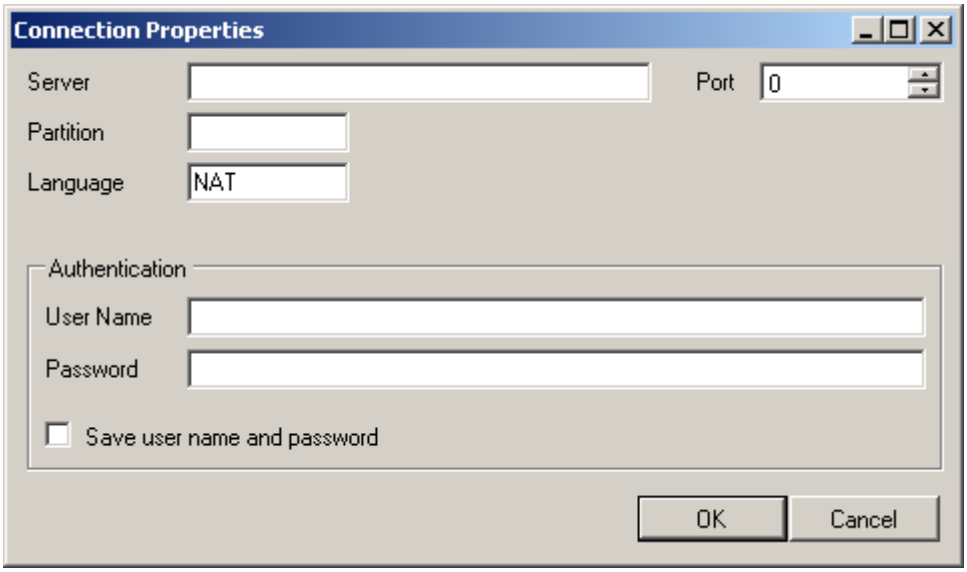

It will appear floating the first time it shows, dock it anywhere you like.



## LANSA Repository Explorer



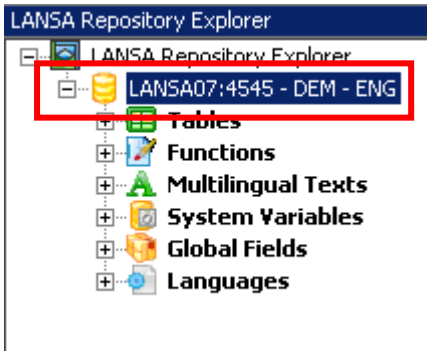

Start by adding a new connection to the Repository Explorer. You can do this either by clicking on the **New Connection** button in the bottom toolbar or by opening up the context menu (right-mouse click), then selecting the **New Connection** menu item.




The **New Connection** dialog box will appear, fill in your connection details then press the OK button. If you are not involved in the administration of a LANSA server, you need to get the following details from your LANSA administrator:

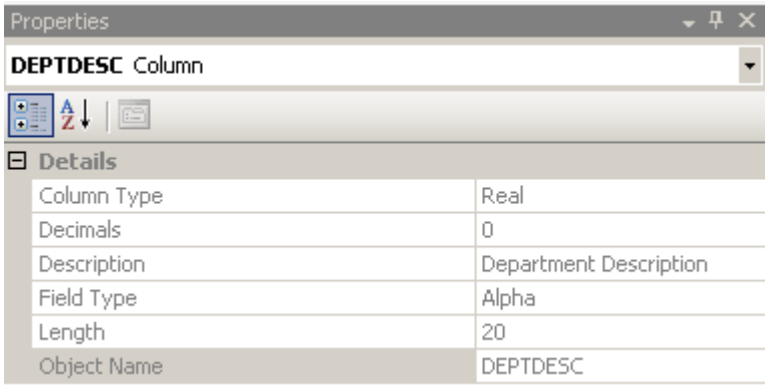
| | |
|---|---|
| **Server** | The name or IP address of the server where a LANSA listener is running. |
| **Port** | The port number that a LANSA listener is running on. One server can have multiple LANSA listeners running on different ports. The default port is 4545. |

| | |
|---|---|
| **Partition** | A LANSA partition you are connecting to. There might be different partitions assigned for development, testing, and production. |
| **Language** | The LANSA language code for this connection. A LANSA language code is a three or four-letter code that indicates a language (for example, ENG represents English language). A LANSA partition can be defined as language-neutral or multilingual (supports various languages). Multilingual partitions allow the storage of field descriptions (among other things) in various languages. |
| **Username and password** | A LANSA username and password for the connection. |

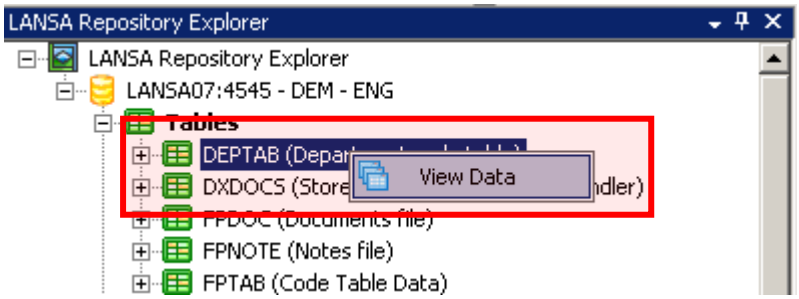



You now have one connection item in your **LANSA Repository Explorer**.
Try expanding the node. It should now attempt to connect to the LANSA server using the connection details you specified. If it connects successfully, it will show **Tables**, **Functions**, **Multilingual Texts**, **System Variables**, **Global Fields** and **Languages**.

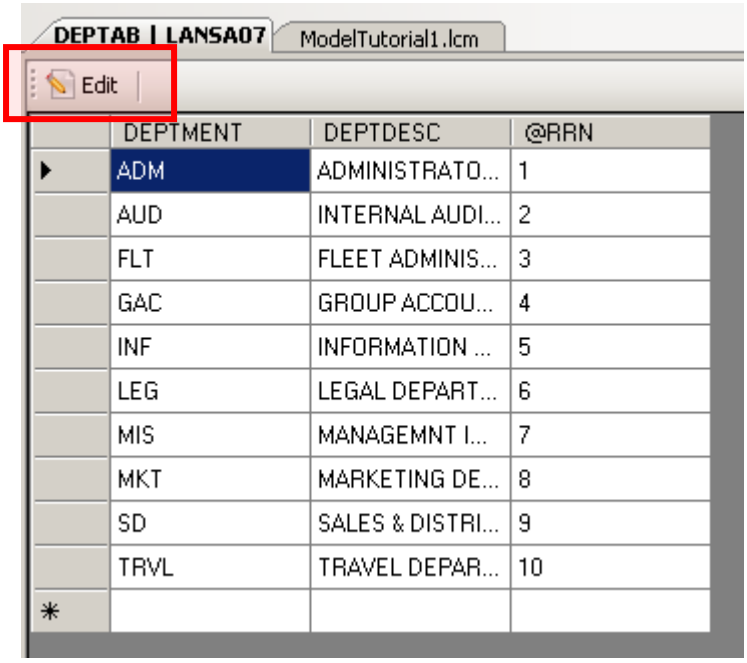
Play around and expand those groups. When you click on an object, the details of the objects will be displayed in the Visual Studio property grid. For example, expand the **Tables** node, look for the table **DEPTAB** (one of the LANSA demonstration objects). Expand the **Columns** node and then click on the **DEPTDESC** node. Now check out your Visual Studio property grid (if you can't see your property grid it might be hidden), it should show the details of the **DEPTDESC** column.




Another handy feature of the LANSA Repository Explorer is its ability to view and edit records in a table. As an example, go back to the **DEPTAB** table node. Notice that on the toolbar (or context menu) you can see the **View Data** button.

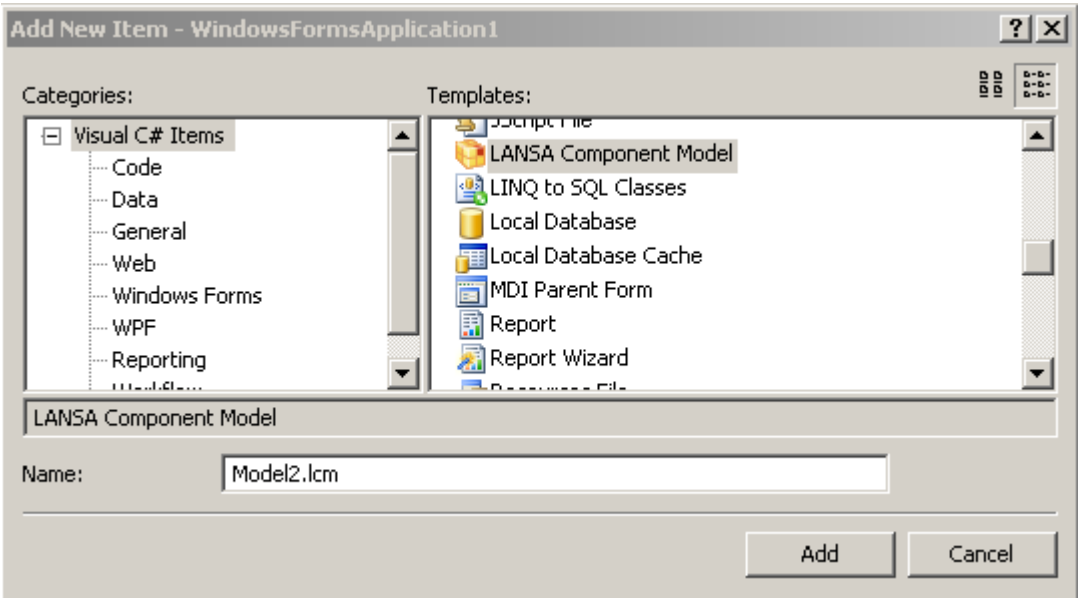


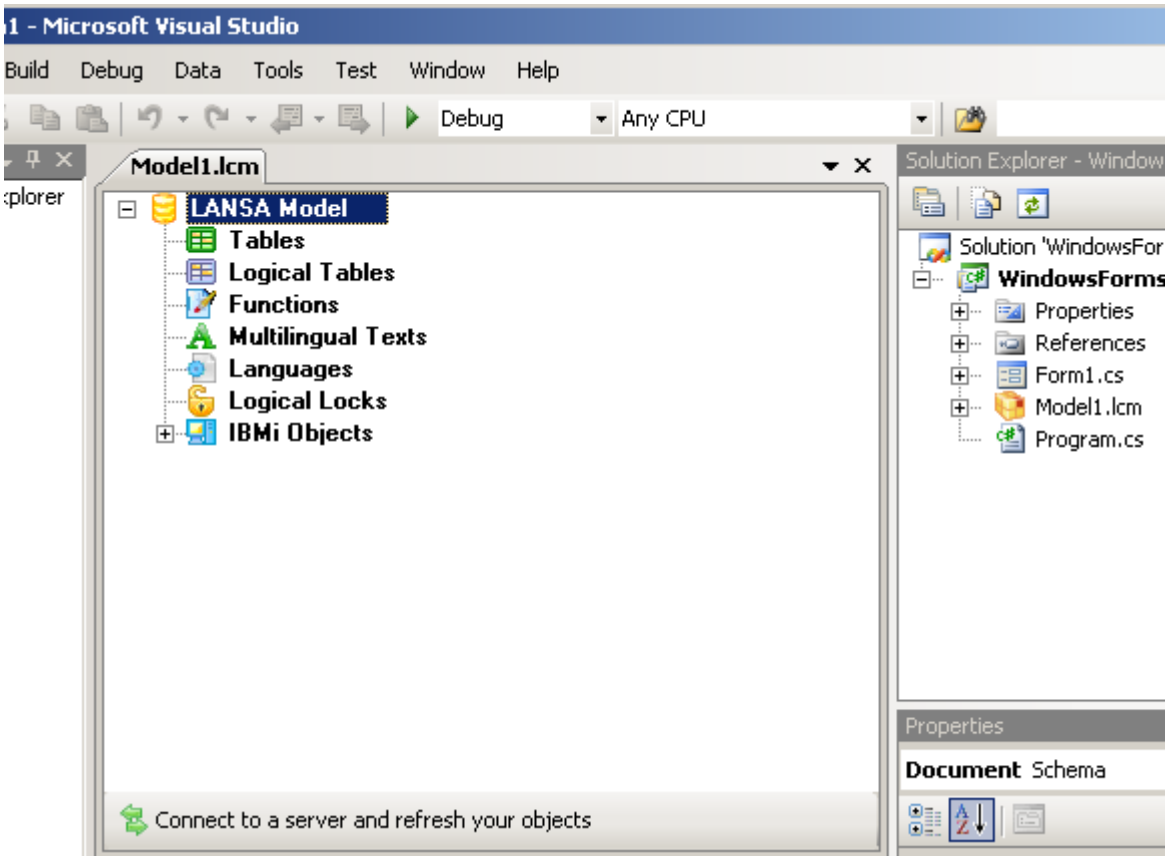

You can now browse the records in the table and edit them if you wish (notice the **Edit** button at the top).




## How Do I Create a New LANSA Data Model File In My Project?

Open the **Add New Item** dialog box and choose the **LANSA Component Model** under **Templates**.






You can now start dragging tables, functions, and multilingual variables from the LANSA Repository Explorer and drop them on the data model editor.

## Can We Add A Data Model (.lcm) to an ASP.NET Web Site?

We can only a data model file to an ASP.NET project, but **not** to an ASP.NET web site.

ASP.NET project is just like any other project and will be compiled into an assembly (DLL).

An ASP.NET web site is more like classic ASP in the sense that:

1. There is no project file that indicates which files should be compiled – all files under the application directory will be included.
2. ASP.NET web sites are compiled on-demand automatically - there is no need to compile an ASP.NET web site before running it.

If you'd like to use an ASP.NET web site instead of an ASP.NET project, you would need to create a separate class library to host your data model file. Your web site will then have to reference this class library in order to access the data model classes.

## Can I Add a Data Model File to Any Project (Languages)?

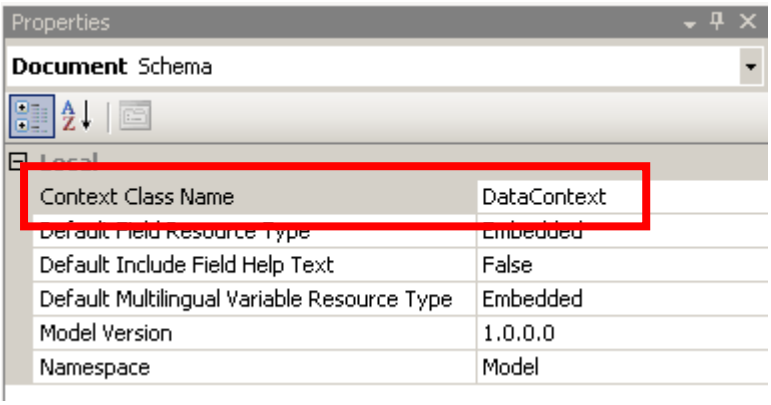
You can only add a data model file to C# and VB.NET projects.

## What If I'm Not Using C# or VB.NET?

If you are developing .NET applications in other languages, you need to create either a C# or VB.NET project in your solution to host the data model file. Your other projects should then reference the C# or VB.NET project containing the data model.

## What Do I Do with the Data Model and How Can I Reference It from My Code?

Every time you save your data model, Visual Studio will automatically generate the classes and methods that represent the objects (and supported operations) in the model. Your code will make use of these classes and methods to perform necessary operations such as retrieving records and executing server functions.
The single point of access to the objects in the data model is the context object. By default, the class will be named **DataContext** (unless you change it in the model properties as illustrated below).




We start always by instantiating the context.

```
DataContext masterContext = new DataContext(false);
```

Think of a context an in-memory representation of the database. Records that are retrieved from the server will become data objects in the client machine's memory and these data objects are managed by the context. When a context is **updatable**, it will automatically keep track of the changes made to any data objects (and also when we create or delete data objects). Remember though that any changes we make to the data objects are happening in memory only and will not be committed to the database until we ask the context to do so by invoking the **SubmitChanges** method. Once **SubmitChanges** is called, any changes we made to the data objects in the context will be committed to the database on the server.

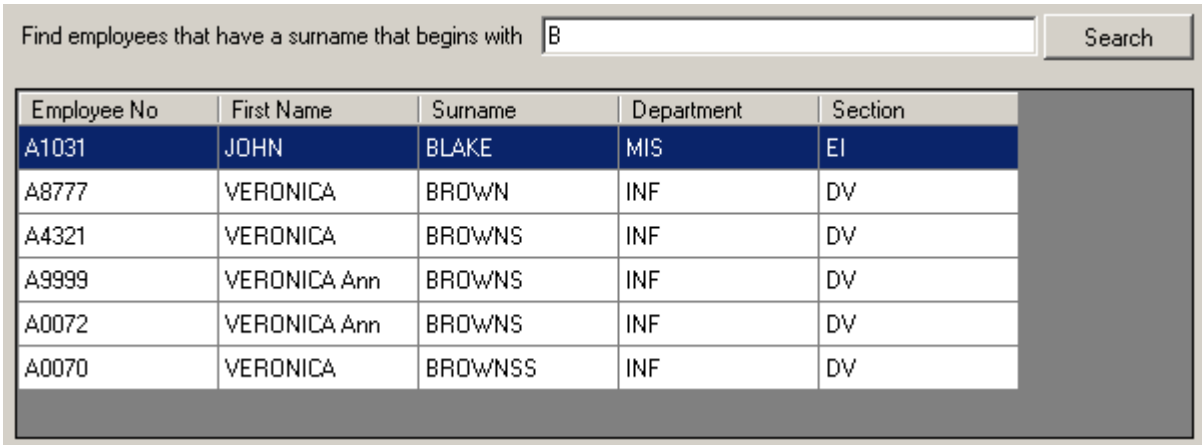
There are two types of context: master (main) context and child context. A context can also be **updatable** or **read-only**. Generally an application would have one instance of master context that will last for as long as the application is running. A master context would generally also be created as a **non-updatable** context.

When your application needs to retrieve some records off the server and it does **not** have any intention of modifying the data, the master context will generally be used to retrieve those records. For example, when your application needs to retrieve a list of available departments in an organization to populate a dropdown, it should use the master context to retrieve those departments.

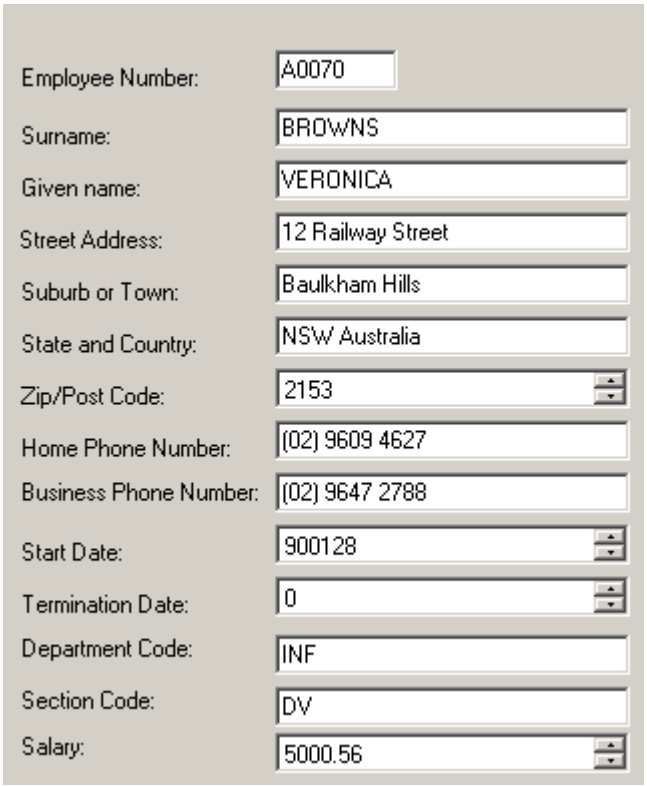
Child contexts, on the other hand, represent **units of work** that involve updating the database on the server. They are created whenever a unit of work starts (generally when a record or a set of records are retrieved from the server) and disposed when the work ends (generally when the record is not required anymore).
Note that a child context is lightweight and not expensive to create. There is no performance penalty in repeatedly creating and disposing child contexts.

A typical use of child contexts would be the retrieval (and update) of information on a **data entry screen**. The following scenario illustrates the common use of child contexts. Imagine that a user of a HR application needs to update some details of an employee.

- The application displays the list of employees (**master context** should be used to retrieve the list).



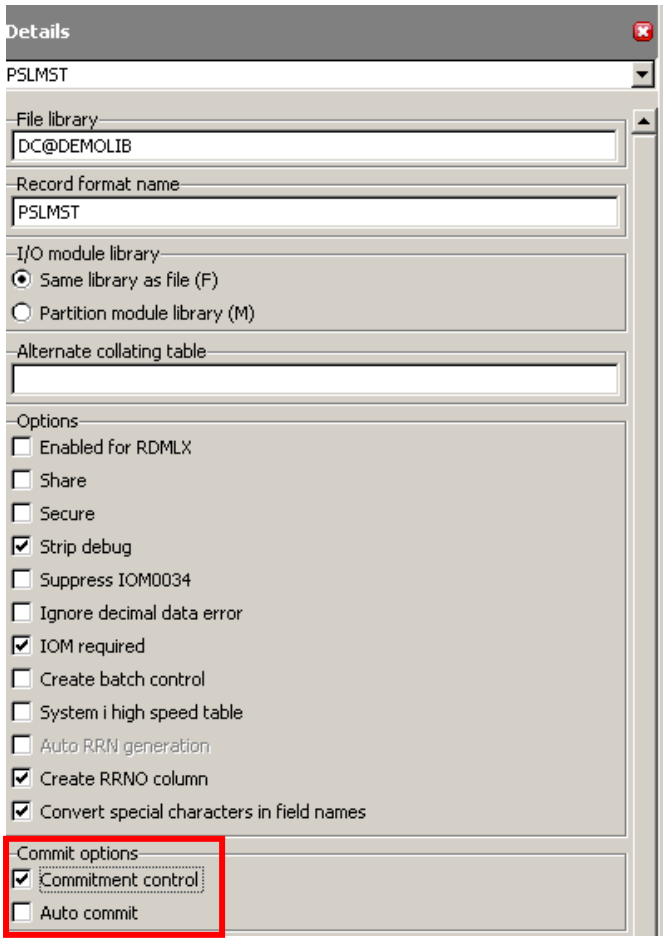

- After the user selects the target employee, the application should then display the employee's details on a data entry screen. To achieve that, the application should create a new child context and use it to retrieve the employee record. A new child context should be created the retrieved employee record can be updated by the user, which means that we need an updatable context to handle this record.



- The record will be retrieved from the server and made available locally in the context as a data object. The application binds the data object with some UI controls to display the information and allow the user to modify some fields. The context will automatically track any changes made to the data object.
- When the user presses the 'Save' button, the application simply needs to call the **SubmitChanges** method of the context to seamlessly apply the changes made in the application to the database on the LANSA server.
- This current child context will be discarded when the user retrieves a different record from the server (and a new child context should be created and used to retrieve the record).

## Enabling Commitment Control on the Server

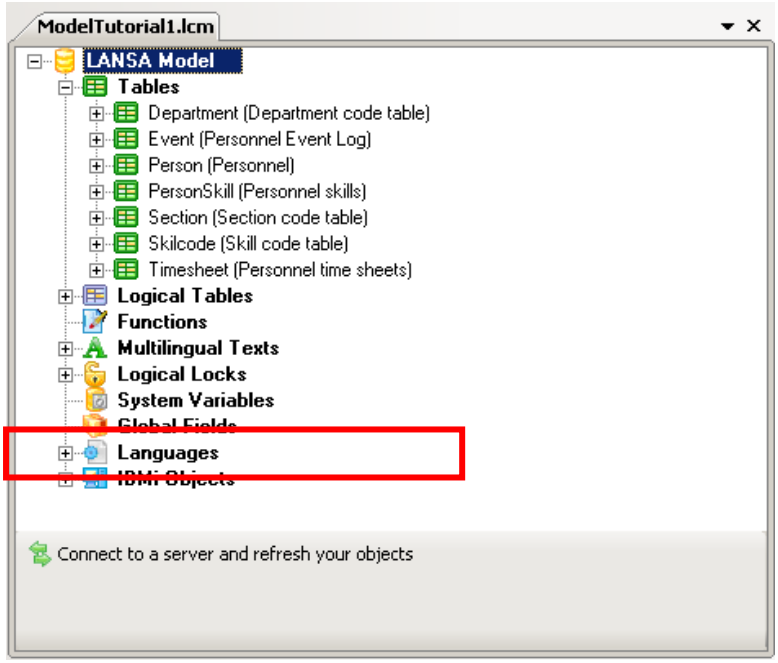
To ensure the integrity of the data when performing updates, it is important that all LANSA tables involved in the updates have their **commitment control** turned on and **auto commit** turned off. When the **commitment control** for a table is turned on, the table will participate in transactions (changes to the table can be rolled-back). These settings can be found on the **File Attributes** tab on the **Visual LANSA IDE**.




## Synchronising Objects on the Data Model with the LANSA Repository

If the structure of an object is changed in the LANSA repository, you need to synchronise your data model with the LANSA repository. For example, if a new column INITIAL is added to the table PERSONNEL in the LANSA repository on the server and you have this table in your data model, you need to synchronise your data model.

To do this use the **Connect to a server and refresh your objects** button as illustrated in the following screenshot (note that you need to have the LANSA Model node as the active node in order to see the button):




## What LANSA Open for .NET Assemblies I need to include when deploying my application?

You need to include only **EntityModel.dll**, which can be found in the **bin** folder of your project, along with your other assemblies (after your project has been compiled).

## Establishing a Connection with a LANSA Server

```
// Specify most details in the Settings object
masterContext.Settings.Server = "LANSA07";
masterContext.Settings.Partition = "DEM";
masterContext.Settings.PortNumber = 4545;

// Connect
masterContext.Connect("username", "password");

// We can also specify all settings (except language) in the Connect method
context.Connect("LANSA07", 4545, "DEM", "username", "password");
```

## Creating a Master Context

A master context is generally read-only (non-updatable). This is indicated by the ***false*** parameter passed to the constructor.

```
DataContext masterContext = new DataContext(false);
```

## Creating a Child Context

```
// The first parameter is the parent context (master context).
// The second parameter 'true' indicates that this context is updatable.
DataContext childContext1 = new DataContext(masterContext, true);
```

## Retrieving Records

Assume that we have a table called ***Person*** in our data model and its plural name is ***Persons*** that maps to the table **PSLMST** which is part of the standard LANSA demo.

```
Person[] personList = masterContext.Persons.RetrieveList();
```

The code above will retrieve all records in the PSLMST table (all columns will be retrieved).
If we want to retrieve only the **FirstName** (GIVENAME) and **Surname** (SURNAME) columns, here is how we do it:

```
Person[] personList = masterContext.Persons.RetrieveList(PersonCols.FirstName + PersonCols.Surname);
```

When we want to retrieve only persons whose salary is greater than 50,000:

```
Person[] personList = masterContext.Persons.RetrieveList(PersonExprs.Salary > 50000);
```

Note how columns of the ***Person*** table are referenced through ***PersonExprs*** (as opposed to ***PersonCols***) when used in an expression.
***PersonExprs*** includes only real columns, whereas ***PersonCols*** includes real columns and virtual columns. (virtual columns are columns whose values are derived from real columns). The implication of this is that you can only use real columns but not virtual columns in your conditional expression.

If we want to retrieve only persons whose salary is between 50,000 and 100,000:

```
var condition = (PersonExprs.Salary > 50000).And(PersonExprs.Salary < 100000);
Person[] personList = masterContext.Persons.RetrieveList(condition);
```

Notice the use of the ‘And’ operator. The ‘Or’ operator can be used in the same way.

We can also search for records that match certain keys. For example, the key column of the PSLMST table is **EMPNO** (assume local name is **EmployeeNo**). The **RetrieveList** methods will look like this:

```
RetrieveList(bool exactMatch, string EmployeeNo)
```

```

```

The *exactMatch* parameter works like this:

- If set to **true**, the key values have to match exactly,
- If set to **false**, partial matches are accepted.

For example, if *exactMatch* is false, a search for EmployeeNo = "A1" will match records with EmployeeNo A1001, A1002, and A1003.

```
// Exact match: the following will return only employee with EmployeeNo = "A1001"
Person[] personList1 = masterContext.Persons.RetrieveList(true, "A1001");

// Partial match: the following will return employees whose EmployeeNo starts with "A1"
Person[] personList2 = masterContext.Persons.RetrieveList(false, "A1");
```

Use the *RetrieveItem* method to retrieve just the first item. Note however that *RetrieveItem* supports only exact matching.

```
// Retrieve the first person whose employee no is "A1001".
// Notice that we don't need to specify exact/partial matching as it's always
// exact matching.
Person person = masterContext.Persons.RetrieveItem("A1001");

// Check for null reference (null is returned if the record can't be found)
if(person != null)
{
  // ...
}
```

## Submitting Changes Back to the Server

Remember that any changes made to the data objects will not be committed to the server until *SubmitChanges* is invoked.

```
context.SubmitChanges();
```

**SubmitChanges** does three things:

- Updates existing records that correspond to modified data objects in the context.
- Creates new records that correspond to newly instantiated data objects in the context.
- Deletes records that correspond to deleted data objects in the context.

## Inserting a New Record to a Table

```
// Create a new 'Person' data object
Person person = new Person();
```

```
person.EmployeeNo = "A9001";
person.FirstName = "Andrew";
person.Surname = "Johnson";

// The context needs to be aware of this new person data object,
// so that later when we ask the context to submit all changes
// to the database, the new person will be included.
// So we attach the new data object to the context.
context.Attach(person);

// At this stage, the person data object exists locally in the context.
// To actually create a new person record on the server,
// we need to call SubmitChanges.
context.SubmitChanges();
```

## Populating Default values of Fields in a Data Object

When we create a new data object, most of the time we want to populate the fields in the data object with their default values (fields have their default values defined in the LANSA Repository). For example, an employee's start date field might have a default value of today's date.

```
Person person = new Person();

// Populate the default values of fields in the person data object
context.SetDefaultValues(person);
```

Please note that **SetDefaultValues** requires an online data context (that is, it has to be connected to a LANSA server).

## Deleting a Record

```
// Indicate to the context that the specified person should be deleted
// the next time SubmitChanges is called
context.Delete(person);

// At this stage, the person record has been marked for deletion.
// We need to call SubmitChanges to actually delete the record on the server.
context.SubmitChanges();
```

## Checking the State of a Data Object

A data object can have one of these 6 states:

| State | Context Type | Description |
|---|---|---|
| Detached | N/A | The data object has not been attached to any context. |

| | | |
|---|---|---|
| | | A data object that has just been instantiated will have this state. |
| New | Updatable | The data object is attached to an updatable context and it does not correspond to any record yet on the database on the server. When SubmitChanges is called, a new record will be created. |
| Unmodified | Updatable | The data object corresponds to an existing record on the database and no changes have been made to it locally. |
| Modified | Updatable | The data object corresponds to an existing record on the database and some changes have been made to it locally (but the changes won't be made to the database on the server until SubmitChanges is called). |
| Deleted | Updatable | The data object corresponds to an existing record on the database and it has been marked for deletion. When SubmitChanges is called, the actual record on the server will be deleted. |
| Readonly | Non-updatable | The data object is attached to a non-updatable context. This is the only state that non-updatable data objects can have. |

To get the state of a context, use the ***GetObjectState*** method of the context instance:

```
Employee employee = new Employee();
DataObjectState state = context.GetObjectState(employee);
```

## Submitting Changes to the Server, Catch and Display Messages when Errors Occur

```
try
{
     context.SubmitChanges();
}
catch (ApplyChangesException exception)
{
     StringBuilder messageBuilder = new StringBuilder();

     // iterate through each error encountered
     foreach (DataObjectErrorInfo error in exception.Errors)
     {
         string message = error.Message;
         messageBuilder.AppendLine(message);
```

```
    }

    // show the error
    MessageBox.Show(messageBuilder.ToString());
}
```

## Submitting Changes to the Server & Resolving Update-conflict

A record encounters an update conflict if after we read the record off the server and before we have a chance to submit our changes back to the server, either:

- Somebody else deleted the record on the server.
- Somebody else has changed some values that we are also changing (for example, we are changing the first name field and the other party is also changing the first name field).

An update conflict needs to be resolved before another attempt is made to submit our changes to the server.

```
// We need to repeat the call to SubmitChanges when conflict has been resolved.
// Use do-while construct to repeat when necessary.
bool repeat;
do
{
    // Do not repeat SubmitChanges if there is no conflict
    repeat = false;

    // Apply changes to the server
    try
    {
        context.SubmitChanges();
    }
    catch (ApplyChangesException exception)
    {
        // Each ObjectChangeConflict represents a conflict information for a record
        foreach (ObjectChangeConflict changeConflict in exception.ChangeConflicts)
        {
            // Handle the 'Person' record
            if (changeConflict.Object is Person)
            {
                Person person = (Person)changeConflict.Object;

                // Check if the person has been deleted from the database?
                if (changeConflict.IsDeleted)
                {
                    // Inform the user that the person no longer exists on the server
                    // Nothing else can be done after that
                    string message = "Employee {0} has been deleted from the server.";
                    message = string.Format(message, person.EmployeeNo);
                    MessageBox.Show(message);
                }
                else
                {
                    string message =
                        "Employee {0} has been modified by somebody else, " +
                        "Keep your changes and discard the other changes?";
```

```
                    message = string.Format(message, person.EmployeeNo);

                    // Ask the user which values to keep
                    var buttons = MessageBoxButtons.YesNo;
                    DialogResult answer = MessageBox.Show(message, "Update Conflict", buttons);
                    if (answer == DialogResult.Yes)
                    {
                        // Keep our changes and discard other person's changes on the database
                        changeConflict.Resolve(ConflictResolveMode.KeepCurrentValues);
                    }
                    else
                    {
                        // Discard our changes and keep the other person's changes on the database
                        changeConflict.Resolve(ConflictResolveMode.OverwriteCurrentValues);
                    }
                    // Must repeat the submit changes after conflict has been resolved
                    repeat = true;
                }
            }
        }
    }
} while (repeat);
```

## Getting the Server to Validate the Changes Made to Records Without Actually Committing the Changes

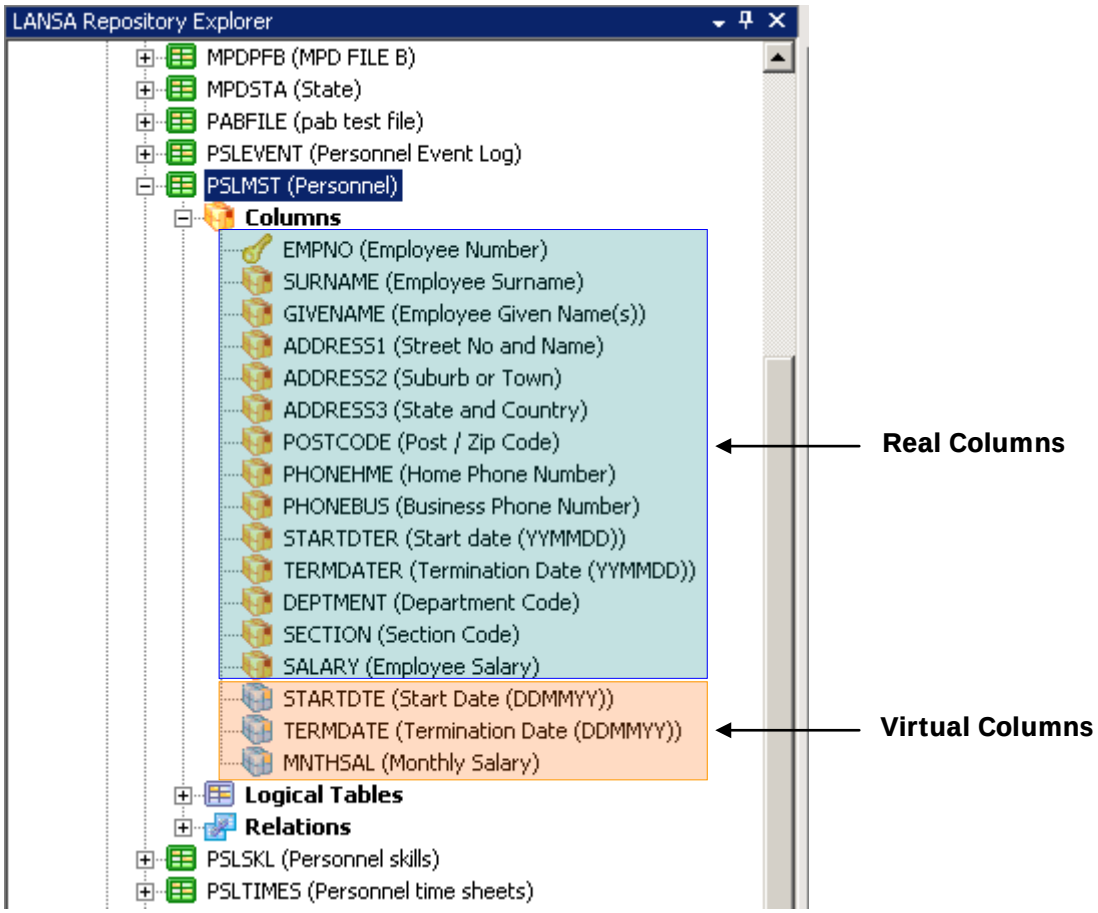
Use the *ValidateChanges* method of the context object

```
DataValidationResult result = context.ValidateChanges();

// Check if validation is successful or not?
if(!result.IsSuccessful)
{
         // "InvalidObjects" property gives us the list of each data object that fails the validation.
         foreach (DataObjectErrorInfo errorInfo in result.InvalidObjects)
         {
               // The 'Columns' property of the error info tells us which columns
               // contain invalid values
               string message = "The following fields contain invalid values: ";
               foreach (Column column in errorInfo.Columns)
               {
                     message += "\n" + column.Label;
               }

               // The 'Errors' property gives us the list of validation messages
               message += "Error messages:";
               foreach (Error error in errorInfo.Errors)
               {
                     message += "\n" + error.Message;
               }

               //
               MessageBox.Show(message);
```

```
        }
}
```

## Real & Virtual Table Columns

A LANSA table's columns can be either **real** or **virtual**.




A **real** column is a column that physically exists on the table and stores a value. A **virtual** column does not physically exist on the table, and its value is derived or calculated from other columns in the table.

## Accessing a Column's Multilingual Texts (Labels, Headings)

There are 2 ways of referencing a column of a table. Let's use the ***Person*** table and ***EmployeeNo*** column as an example.

The first way to reference the EmployeeNo column is through the ***PersonCols*** class (context-less column):

```
var employeeNoCol = PersonCols.EmployeeNo;
```

The second way to reference the EmployeeNo column is through the context's ***Persons*** entity (context-aware column):

```
var employeeNoCol = context.Persons.Columns.EmployeeNo;
```

It's fine to use either way if we just want to specify which columns to retrieve when requesting some data off the server. However if we want to access the multilingual texts of the column we have to use the context-aware columns (since the texts returned will depend on the language currently used).
So if we try to do the following, an exception will be thrown.

```
string label = PersonCols.EmployeeNo.Label;
```

The right way to get the label of the column is one of the following two ways:

```
string label = context.Persons.Columns.EmployeeNo.Label;
string label = context.Localized[PersonCols.EmployeeNo].Label;
```

## Deleting Multiple Records

To delete several records at once, use the ***DirectDelete*** method.
**Important note**: ***DirectDelete*** operates directly on the database on the server – records are deleted right away, not when SubmitChanges is called (no call to SubmitChanges is required).

To delete all employees in AUD department:

```
context.By_DeptEmployees.DirectDelete("AUD");
```

To delete all employees with salary greater than 50,000:

```
context.Employees.DirectDelete(EmployeeExprs.Salary > 50000);
```

## Updating Multiple Records

To update several records at once with the same values, use the ***DirectUpdate*** method.
**Important note**: ***DirectUpdate*** operates directly on the database on the server – records are updated right away, not when SubmitChanges is called (no call to SubmitChanges is required).
Unlike ***DirectDelete***, ***DirectUpdate*** does not support expressions.

To update the salary of all employees in AUD department to 50,000:

```
Employee values = new Employee();
values.Salary = 50000;
context.Employees.DirectUpdate(values, EmployeeCols.Salary, "AUD");
```

## Beginning & Ending a Transaction

Transaction can be started by invoking the **BeginTransaction** method of the context object.

```
```

```
context.BeginTransaction();
```

Ending a transaction is done by invoking either **CommitTransaction** or **RollbackTransaction**.

```
// Commit
context.CommitTransaction();
// Rollback
context.RollbackTransaction();
```

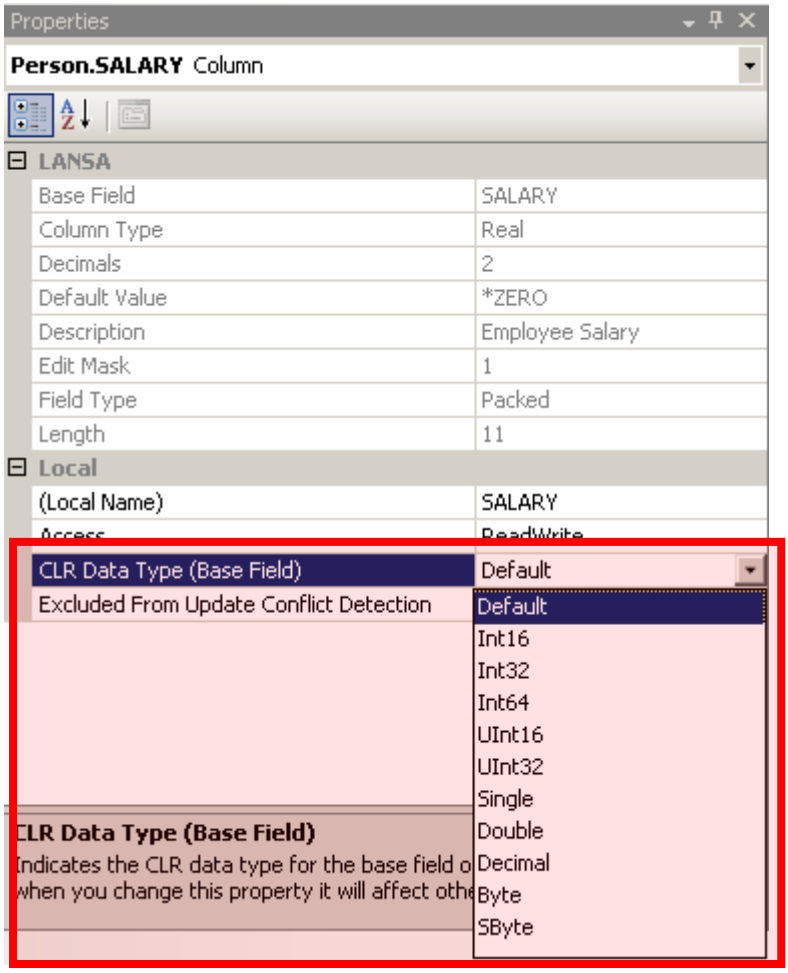
Notes:

- **SubmitChanges** method automatically starts a transaction before it begins applying changes to the database. If no error occurs, it will automatically **commit** the changes, otherwise a **rollback** will be performed.
- In order for a database table to support this transaction, it needs to have its **commitment control** turned on. Please see the section **Enabling Commitment Control on the Server** for further details.

## How Do the LANSA Field Types Map to .NET Runtime Types?

| LANSA Field Type | .NET Runtime Type | Overridable in the Model? |
|---|---|---|
| Signed<br>Packed<br>(where decimals > 0) | Decimal | Yes |
| Signed<br>Packed<br>(where decimals = 0) | Int64 | Yes |
| Integer | Int64 | Yes |
| Float | Double | Yes |
| String<br>Alpha<br>Nchar<br>NVarChar<br>CLOB<br>BLOB | String | No |
| Date<br>Time<br>DateTime | DateTime | No |
| Binary<br>Varbinary | Byte[] | No |
| Boolean | Bool | No |

The .NET run time data type mappings for numeric fields can be overridden in the model by selecting column and changing its **CLR Data Type (Base Field)** property:




## Data Context Advanced Options

The properties that control the behaviour of the data context reside in the **Options** object in the data context. Code example to change the property `PerformBasicDataValidation`:

```
DataContext context = new DataContext(true);
context.Options.PerformBasicDataValidation = false;
```

Below is the list of the available options and their descriptions.

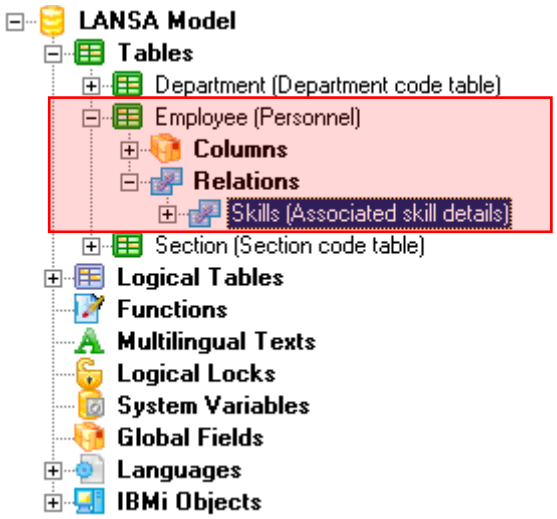
| | |
|---|---|
| RowIdentityType | Indicates the identity system to use to identify data rows.<br>Data rows can be identified using either their Relative Record Numbers (RRN) or their primary keys.<br><br>**Relative record number**: a relative record number identifies the positions of the records relative to the beginning of the file. For example, the relative record numbers of the first, fifth, and |

| | |
|---|---|
| | seventh records are 1, 5, and 7, respectively<br><br>Generally speaking, RRNs should be used whenever they are available, however there are circumstances when it is recommended to use primary keys instead of RRNs. An example would be when providing an offline data entry support. In this case, a context or data object is serialized and then stored for an extended period of time on the user's local computer harddrive. Since an RRN of a record in an IBM i database table is not guaranteed to remain the same (reorganization of the tables might change RRNs), it becomes necessary to identify records using their primary keys instead.<br><br>In an IBM i system, all tables will have RRN. This is however not the case in a Microsoft Windows system where the existence of an RRN column in a table is an option that can be set by the creator of the table (through Visual LANSA IDE).<br><br>**RowIdentityType** can be set to either:<br>• `UseRelativeRecordNumberWhenAvailable`<br>The Relative Record Number (RRN) should always be used as a row's identity when the table support RRN, otherwise use the table primary keys.<br>• `AlwaysUsePrimaryKey`<br>Primary keys should always be used as identity regardless of whether the table supports RRN or not |
| PerformBasicDataValidation | Indicates whether basic data validations should be performed on the client before data is submitted to the server.<br>Basic data validations are simple and automated validations that are performed based on each column's input attributes and data type as defined in the LANSA repository.<br>Note that custom validation rules defined in the repository are not part of the basic data validation (custom validation rules are imposed on the server, not on the client).<br><br>**About Input Attributes**<br>Any LANSA field defined in the repository has a set of **input attributes** that constraint the kind of value that the field can contain. For example, we can indicate that a SURNAME field should always have a value. |

| | |
|---|---|
| | **PerformBasicDataValidation** can be set to either **true** or **false**.<br>The list of checks that are performed in a basic data validation are as follows:<br>• If the field does not allow blank value, check that the value of the field is not blank.<br>• If the field is marked as containing an IBM i object name, check that the value of the field is a valid IBM i object name.<br>• If it is indicated that the value of the field should occupy the entire field length, check that the length of the value is the same as the maximum length of the field.<br>• If the field is marked as unsigned, check that the value is not negative.<br>• Check that the length of the value does not exceed the maximum length of the field.<br>• Check that the number of decimal digits does not exceed the maximum allowed. |
| PerformServerValidationOnSubmitChanges | Indicates whether data validation on the server should be performed on **all** changed records as a whole **before** modifying the records on the database.<br>Note that server validation will always be performed regardless of whether this property is set to **true** or **false**, it's just a matter of **when** the validation will be performed.<br>To illustrate the difference better, imagine that we have 2 records to update. Notice the sequence of events for each case.<br>**With PerformServerValidationOnSubmitChanges property set to false**:<br>1. Validation of record 1<br>2. Update record 1 on the database<br>3. Validation of record 2<br>4. Update record 2 on the database<br>If step 3 fails, step 2 would have been performed which means record 1 would have been updated on the database. If the particular table does not support **commitment control**, the runtime does not have any way to rollback the changes made in the step 2 and the integrity of the database could be compromised.<br>**With PerformServerValidationOnSubmitChanges property** |

<table>
<tr><td></td><td><u>set to true</u>:<br>1. Validation of record 1<br>2. Validation of record 2<br>3. Update record 1 on the database<br>4. Update record 2 on the database<br>If step 2 fails, no records will have been updated so the integrity of the database has not been compromised.<br><br>Enabling PerformServerValidationOnSubmitChanges might have performance impact, however it should be negligible on most cases and it is recommended that you enable this property at all times.</td></tr>
<tr><td>CheckUpdateConflicts</td><td>Indicates whether to detect if update conflict is present when trying to update a record.<br>An update conflict is present if the runtime finds that the record was updated by another entity (another instance of the application for example) between the time the record was retrieved and the time the update request was made.<br><br>If CheckUpdateConflicts is set to true, the runtime will detect if such conflicts are present and raise an exception if conflicts are found.<br>Conflicts must be resolved (either by discarding the current value in the memory or the current value in the database) before another attempt at the update can be made.</td></tr>
<tr><td>UpdateModifiedColumnsOnly</td><td>Indicates whether only columns (and by implications objects) that are modified (in memory) should be submitted for update. For most cases, it is sufficient and recommended to update only modified columns, however there might be times when you might prefer to update all columns (set this property to false).<br><br>For example, we retrieve a record from a table called Person<br><br>Person person = context.Persons.RetrieveItem(…)<br><br>We change only the Salary column & call SubmitChanges (UpdateModifiedColumnsOnly is true)</td></tr>
</table>

| | |
|---|---|
| | `Person.Salary = Person.Salary * 1.05;`<br>`context.SubmitChanges();` |
| | Since **UpdateModifiedColumnsOnly** is **true**, the runtime will try to update only the **Salary** field and completely ignore other fields. This means that update-conflict check will only be performed on that field.<br>If in the meantime, somebody else changes say the Postcode field of that person, you will not be aware of it as no update-conflict will be raised.<br><br>Please note that setting **UpdateModifiedColumnsOnly** to **false** could carry a performance penalty on updating records, especially if there are many records in the data context (remember that the runtime will have to go through all records in the data context even for those that have not been modified since the runtime has to compare the in-memory values with actual values of those records in the database).<br>It is therefore important to make sure that you dispose or clear your data context whenever you are retrieving a new set of records to ensure that only records that are relevant to the current unit of work are present in the data context. |
| DataSource | Indicates where data is coming from when retrieving data (using **RetrieveList** or **RetrieveItem** operations).<br><br>There are 2 possible values:<br>• **Database**<br>The records will be retrieved from the database on the server.<br>• **Dataset**<br>The records will be retrieved from the context's embedded dataset. A context's embedded dataset is accessible through the DataSet property of the data context (e.g. context.DataSet).<br><br>The **DataSource** property is useful when your application supports offline mode. This way you can use the same fragment of code to retrieve your records for both offline & online mode, you just need to set the **DataSource** property accordingly (**Database** for online, **Dataset** for offline). |

| | |
|---|---|
| ListAutoLoadEnabled | Indicates whether child list should automatically be populated when any operation is performed on the list for the first time (for example when the **Count** property is accessed). |
| AttemptUpdateLockingWait | Indicates how long (in milliseconds) the runtime should wait when trying to acquire an update lock for a data row before it gives up and generates an error.<br>The runtime locks a data row before it proceeds with the change conflict detection to ensure that nobody updates the row while the change conflict detection is performed. |
| AutoDeleteDependentRecords | Indicates whether child records should automatically be deleted when the parent record is deleted.<br>Child records are records whose foreign key values are set to the primary key values of the parent record. The parent table has a 1-to-N relationship with the child table.<br>Enable the use of this feature with care.<br><br>To illustrate the use of this property, imagine we have such tables in our data model. Notice that the Employee table has a relationship with the Skills table (1-to-N).<br><br>  <br><br>On the LANSA Repository, there is a validation rule defined that prevents an **Employee** record who has Skill records associated with it from getting deleted.<br>This means that if we want to delete an **Employee** record that has some **Skill** records associated with it, we need to delete the |

| | |
|---|---|
| | **Skill** records first.<br><br>To let the runtime do this automatically for us, we can set **AutoDeleteDependentRecords** to **true**. But note that we also need to set the **PerformServerValidationOnSubmitChanges** to **false**, otherwise the validation phase will always fail since the runtime will have **not** deleted the associated Skill records in the validation phase (the validation phase has no way of knowing that the runtime is intending to delete the Skill records). |

## How to Assign a Database NULL Value to a Column in a Record?

If a column in a database table is nullable (a column is nullable if the LANSA field it is based on has an input attribute ASQN which stands for **A**llow **SQ**L**N**ULL).
If the data type of the column is not alphanumeric, it will map to a value-type CLR data type (such as Int32, Double). Since these types cannot be set to **null**, the LANSA Open for .NET runtime needs to use special values to indicate that we want to set those columns to **database NULL** when saved to the database.
By default the special values representing the **database NULL** are the **MaxValue** of the data type for value-type and **null** for reference type. For example, to indicate that we want to assign a **database NULL** value to the **Person.TerminationDate** (DateTime) column and also to the **Person.Notes** (string) column:

```
Person person = ...
person.TerminationDate = DateTime.MaxValue;
person.Notes = null; // string is reference type so set to null

// When we call SubmitChanges, both the TerminationDate and Notes column of this record
// will be set to NULL on the database
context.SubmitChanges();
```

We can change the special values that indicate **database NULL** to something else. We do that by using the static method **SetDbNullValue** of the **Context** class**.** Example of usage:

```
// To set the database NULL value of Int64 value to Int64.MinValue
Context.SetDbNullValue(typeof(Int64), Int64.MinValue);
```

## Accessing Original Values in a Data Object

Original values are values that are retrieved from the database. To find out the original values, use the OriginalValues property of the data object (it contains properties; each represents a field in the data object).

Code example:

```
```

```
Person person = context.Persons.RetrieveItem(...)

// Modify the Salary field of the person
person.Salary = person.Salary * 1.05;

// Get the original value of the Salary field as retrieved from the database
var originalSalary = person.OriginalValues.Salary;
```

## Getting Error Information From ApplyChangesException

ApplyChangesException is the exception that is thrown when an attempt to update records on the server's database failed (through the call to the data context's SubmitChanges method). An attempt to update records could fail if:

- Basic data validation (client side validation) failed.
- Server-side validation failed.
- Update conflicts are present.
- A fatal error occurs on the server.

An **ApplyChangesException** object has two properties that tell us more about the nature of the problem:

- **Errors** (Collection of **DataObjectErrorInfo**)
  A collection of error information items. Each item is associated with a data object and it contains the details of the error that occurred for that data object.
- **ChangeConflicts** (Collection of **ObjectChangeConflict**)
  A collection of update conflicts. Each item is associated with a data object and contains the details of the update conflicts that are present for that data object.

Each item in the **Errors** collection is a **DataObjectErrorInfo** object, and it has the following properties that describe the error that occurred for a data object:

| | |
|---|---|
| Object | Reference to the data object that the error was caused by. |
| ErrorReason | Indicates the cause of the error. Possible values:<br>• **PrevalidationError**: some values in the data object failed the basic validation (client-side validation).<br>• **ValidationError**: the data object failed the server side validation.<br>• **KeyAlreadyExists**: this error occurs on the insertion of new record. It indicates that there is already another record in the database with the same key values as the new record we are trying to insert. |
| Columns | When the cause of the error is either **PrevalidationError** or **ValidationError**, this property will contain the columns that failed the validation.<br>If the error is caused by **PrevalidationError**, information on the reason the validation failed can be retrieved from the **PrevalidationColumnErrors** property. |
| PrevalidationColumnErrors | When the cause of the error is **PrevalidationError**, each item in this list contains |

| | |
|---|---|
| | the reason the validation failed for the column specified in the **Columns** list on the same index (PrevalidationColumnErrors[i] corresponds to the column specified in Columns[i]). |
| Errors | List of detailed error messages. |
| Message | All error messages as one string (multiline, each error message is placed in a new-line). |

Each item in the **ChangeConflicts** collection is an **ObjectChangeConflict** object, and it has the following properties that describe the update-conflicts that occurred for a data object:

| | |
|---|---|
| Object | Reference to the data object that caused the update conflicts. |
| IsModified | Indicates if the conflict occurs because some columns have been modified on the database since they were last read by this application. |
| IsDeleted | Indicates if the conflict occurs because the actual database row that corresponds to this data object has been deleted. |
| MemberConflicts | List of items, each item provides information on each individual column that caused an update conflict.<br>Each item is a MemberChangeConflict object and contains the following properties:<br>• **Column**: indicates the column that causes this conflict. You can use the properties of the column (such as **Description**) to display meaningful information for the users of the application.<br>• **OriginalValue**: the original value of the column (as retrieved from the server) by this application.<br>• **CurrentValue**: the current in-memory (local) value of the column.<br>• **DatabaseValue**: the current database value of this column on the server's database. |

## Handling Locking of Row before Update

LANSA Open for .NET runtime automatically creates a logical lock for every row that it is about to update to avoid simultaneous update by different parties. The lock is applied before change-conflict detection takes place to guarantee the integrity of the change-conflict detection process. By default, the lock name will be the name of the LANSA table, and the identifiers will be either the Relative Record Number (RRN) or the primary key values.

However, your .NET application might not be the only application that has the ability to update records. Legacy applications (such as LANSA RDML applications) might have already been in operation, and they most likely would have their own locking mechanism. In this case, the locking routine in your .NET application will have to work in the same way as the existing LANSA applications.

To provide your own locking routine, we need to create two event handlers and attach them to these 2 static events:

- DataContext.Events.LockObjectForUpdate
- DataContext.Events.UnlockObjectAfterUpdate

As an example, assume that we have a logical lock defined in our model called ***Employee***. Our locking routine will look like the code below. Note that it is very important to assign the **args.Successful** property with **true** or **false**. If you don't assign this property, LANSA Open for .NET runtime will assume that you do not have your own routine and it will execute its own standard locking routine.

```
private void Events_LockObjectForUpdate(object sender, LockObjectEventArgs args)
{
        if (args.Object is Employee)
        {
                Employee employee = (Employee)args.Object;
                try
                {
                        context.LogicalLocks.Employee.Lock(employee.EmployeeNo);
                        args.Successful = true;
                }
                catch (ObjectAlreadyLockedException)
                {
                        args.Successful = false;
                }
        }
}
```

## Serializing a Context

Serializing a **context** means generating a string representation of the context. Some practical use of context serialization:

- To enable a context to be persisted on any medium. For example, web applications have to persist the context somewhere in-between requests since they are stateless in nature. Another application is to provide offline data entry support (ability to enter data when the application is not connected to a server).
- To transport a context easily to another computer. For example, in an architecture where a client application does not talk directly with a LANSA server, but it talks with an intermediary (application server), which in turns talk with the LANSA server (client → app-server → LANSA). In such architecture, the context will need to be transported back and forth between the client and app-server.

### Serialization

```
// Retrieve a person
Person person = context.Persons.RetrieveItem("A1001");

// Change the salary
person.Salary += 10000;

// Use object reference store to to keep a reference to the person data object,
// otherwise we will lose it after we deserialize the context back.
context.ObjectReferenceStore.Add(person);
```

```
// Serialize the context.
// Notice that we have not submitted the changes back to the server.
// We will do that later after we deserialize the context
// to illustrate that context serialization fully preserves the
// context's state.
string serializedContext = context.Serialize();
```

**Deserialization**

```
// Get the stored string representation (serialized context).
string serializedContext = . . .;

// Reconstruct the context from the string representation of it.
DataContext context = DataContext.CreateFromSerialized(serializedContext);

// Restore our reference to the person data object.
Person person = (Person)context.ObjectReferenceStore[0];

// We might display the values in the person object
// to the user e.g. on a data entry screen.
// ...

// Submit changes back to the database.
// This will include the change made to the salary field before serialization/deserialization.
context.SubmitChanges();
```

Note the use of the **object reference store** to maintain the reference to our **Person** data object. Right before we serialize the context, we add the reference to our **Person** data object to the context's object reference store. When the context is serialized, it will also serialize the reference to our **Person** object. When we deserialize the context back, we can get back the reference to the **Person** object by retrieving it from the object reference store.

## Serializing Data Objects in a Context

We can also serialize some data objects instead of the whole context, which will generate the string representation of those data objects. When deserialized, the data objects can be attached to any existing context.

**Serialization**

```
// Retrieve a person
Person person = context.Persons.RetrieveItem("A1001");

string serializedPerson = context.Serialize(person);
```

**Deserialization**

```
// Get the stored string representation of the Person data object.
```

```
string serializedPerson = . . .;

// Reconstruct the Person from the string representation of it
// The resulting Person data object will be attached to the context
// that performs the deserialization.
Person person = context.DeserializeObject(serializedPerson);
```

You can also serialize a collection of data objects as illustrated below:

```
// Retrieve all persons
Person[] personList = context.Persons.RetrieveList();
string serializedList = context.Serialize(personList);
```

## Turning On LANSA Connection Pooling

When connection pooling is turned on, LANSA Open for .NET will run a pool of connections that are kept open and ready for use when a request is made. The purpose is to avoid having to establish a new physical connection with the LANSA server every time a request comes in (establishing a new connection is a costly operation).
Connection pooling is normally not required for desktop (client) applications, however it generally should be enabled for web applications as web applications are stateless (which means that they are required to open a new connection at the beginning of each request and close it at the end of the request). As mentioned earlier opening a new physical connection takes time. If pooling is not enabled, the webserver response to the browser might be delayed while it is waiting for the connection to the LANSA server being established.

The class that controls all aspects of the connection pooling is **LOpen.EntityModel.ConnectionPool**.

To enable the pooling, set the **Enabled** property to true.

```
ConnectionPool.Enabled = true;
```

Below is a list of the properties that control the behaviour of the connection pooling.

| Property Name | Description |
|---|---|
| `MaxPoolSize` | The maximum number of connections that the pool can have.<br>This is to ensure that the web application does not open too many connections to the LANSA server. |
| `MinPoolSize` | The minimum number of connections that should be maintained in the pool.<br>This is to ensure that response time for most web requests will be consistent. |

| | |
|---|---|
| `InactiveConnectionTimeout` | Indicates how long (in seconds) an unused connection should be maintained in the pool before it can be closed and removed from the pool (assuming that the number of connections in the pool is more than the **MinPoolSize**). |
| `AcquireConnectionTimeout` | Indicates how long (in seconds) the application should wait when no connection is available in the pool (and the number of connections in the pool has reached **MaxPoolSize**). |

When writing ASP.NET applications, the appropriate place to put the code that configures the connection pooling behaviour is the **Application Start** event in the **Global.asax** file as illustrated by the example below:

```
void Application_Start(object sender, EventArgs e)
{
    // Enable the connection pooling
    ConnectionPool.Enabled = true;

    // Setup the connection pooling behaviour
    ConnectionPool.MaxWaitForConnection = 30; // 30 seconds
    ConnectionPool.MaxPoolSize = 10;
    ConnectionPool.MinPoolSize = 5;
    ConnectionPool.InactiveConnectionTimeout = 600; // 10 minutes
}
```

## DataContext Serialisation & Deserialisation

Serialising a DataContext generates a string representation of the current state of the DataContext. The current state of a DataContext includes table rows that have been retrieved from the database and their state (that is which column values have been modified or which rows have been deleted).
This string representation can then be deserialised back to a DataContext object.

**Why do we need to be able to serialise a DataContext into a string?**

There are times when it is not feasible or possible to continuously keep the DataContext in memory as an object. For example, when we are developing an ASP.NET application, so that the DataContext survives between the stateless web requests, we need to save the DataContext somewhere between one request and the next.
We might prefer to keep the DataContext in the “view state” as opposed to using the in-memory **Session** object. Or we might want to store the DataContext in an SQL server table. If we choose to store the DataContext in the “view state” or as a value in an SQL server table, we need to serialize the DataContext first to its string representation, store it in the “view state” or the table, then deserialize it back to the DataContext object on the next request.

**Offline Operation**

DataContext serialisation can also be useful when our application supports offline data entry. For example, a contractor working on a site might not have online connection to the LANSA server. In this case, he/she will need to enter the data offline and later reconcile the new data with the database on the LANSA server. In this case, the new records will be created and stored in the DataContext. The DataContext will then be serialized into a string and stored locally in the contractor’s notebook harddrive. The next time the contractor connects the application to the LANSA server, it will simply deserialize the DataContext containing the new records and

simply call a **SubmitChanges** on the DataContext to reconcile all the new records with the database on the LANSA server.

## Serialising a DataContext

```
string contextAsString = context.Serialize();
```

## Deserialising a DataContext

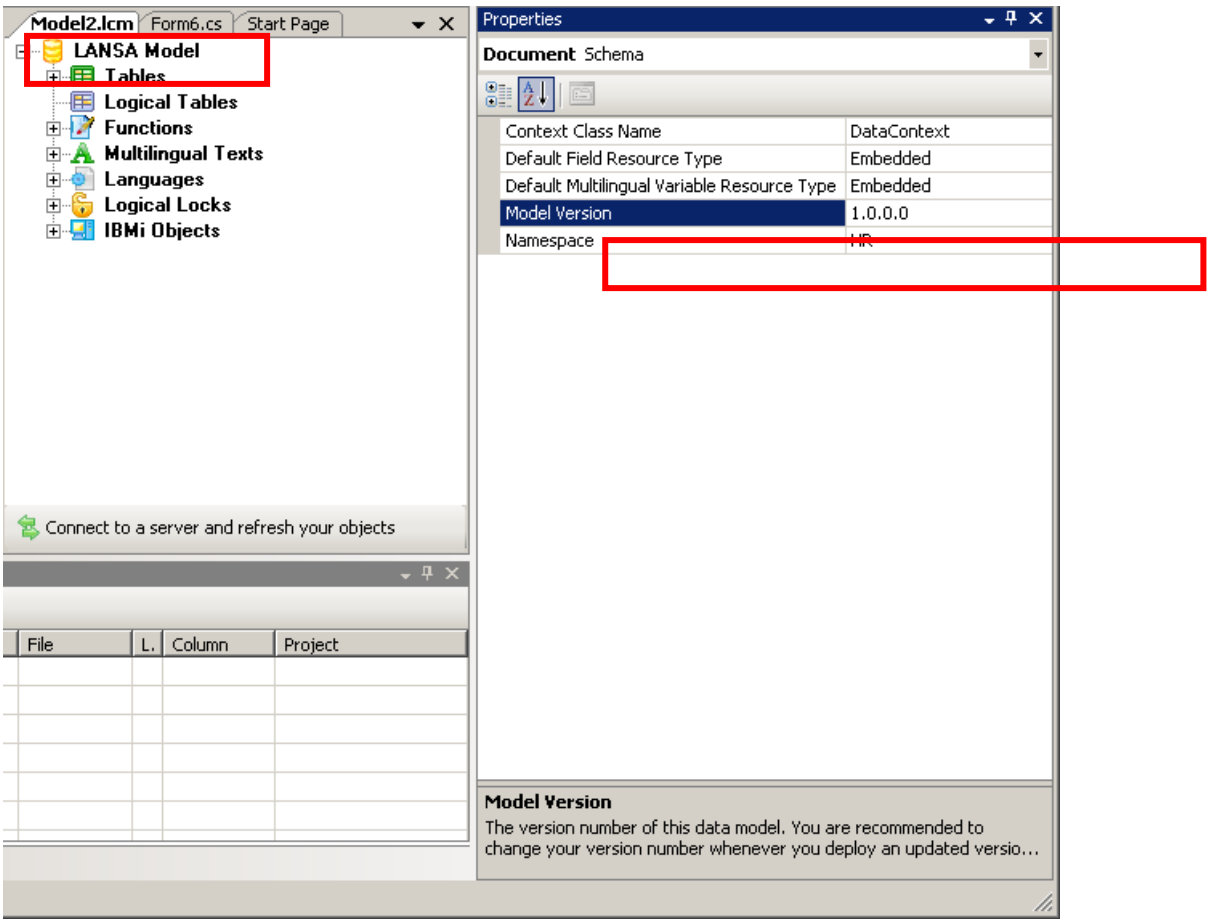
```
DataContext context = DataContext.CreateFromSerialized(contextAsString);
```

Or if we want to use a specific instance of DataContext:

```
// Somewhere we have the DataContext instantiated
context = new DataContext(true);
...
// Now we want to use the same instance when deserialising
// instead of creating a new one
context.Deserialize(contextAsString);
```

## Data Model Version Number

Our LANSA data model has a version number that we can set. The figure below shows where we can set this version number (it's a property of the root node of our data model document).



If we are serialising our DataContext and we store the string representation for an extended period of time (as in the case with the offline data entry capability), we need to make sure that when we update our LANSA data model we change the version number, so that the L/Open.NET runtime can tell if a string representation of a DataContext is compatible with the DataContext that tries to deserialise it. If you do not change your data model version after making a change to it, L/Open.NET runtime will have no way of knowing that your data model has changed. If you then deserialise a string that was serialised with the previous version of the data model, *the data in the DataContext will be corrupted*. So it is very important that we change our data model version number when we change anything in our data model.

**Another Use of DataContext Serialisation**

DataContext serialisation can also be useful when the client application communicates with an application server (using remoting or web services) instead of directly to a LANSA server. The application server communicates directly with the LANSA server. This architecture gives the client application freedom in regards to the way data is transported between the client and the application server. The client can now use web services or .NET remoting to transport data.

In this sort of architecture, we can see where the DataContext serialisation comes in.

1. The client application will have an offline DataContext (a DataContext that is not connected to a LANSA server).
2. When the user requests a record to be retrieved, the client then passes the request to the application server.
3. The application server uses the online DataContext (connected to a LANSA server) to fetch the requested data into its DataContext.
4. The application server then serialises the DataContext and then transmits it to the client application.
5. The client application receives it and deserialises it back to a DataContext object.
6. The user can then modify the value of some fields and the changes will automatically be tracked by the DataContext.
7. When the user requests the changes to be saved, the client application serializes the DataContext & send it to the application server.
8. The application server receives it, deserialises it back to a DataContext object.
9. The application server then calls **SubmitChanges** on the DataContext to get the LANSA server to update its database with the changes.
10. An error might occur in the application server and an **ApplyChangesException** exception might be raised. Since this needs to be passed back to the client application, the application server serialises this exception, transmit to the client who then deserialises it back to an **ApplyChangesException** object.

**Serialising ApplyChangesException Object**

```
catch (ApplyChangesException ex)
{
      string asString = context.Serialize(ex);
}
```

**Deserialising ApplyChangesException Object**

```
ApplyChangesException ex = context.DeserializeApplyChangesException(asString);
```

**DataValidationResult** object can also be serialised and deserialised in a similar manner as above.

## Context's Embedded Dataset

Each instance of a context contains an embedded dataset, which is a container where we can put various data objects in. The reasons why we want to put our data objects in the context's embedded dataset are:

- All data objects stored in the dataset will be preserved when the context is serialized.
- The data objects stored in the dataset can be used as a data source for data retrieval. This means that when we do say a RetrieveList operation for a Person table, the data can come either from:
    1. The database on the server, or
    2. The objects stored in the context's embedded dataset.

To indicate to a context that it should retrieve from its dataset instead of the server, set the **Options.DataSource** property to **Dataset** (by default it is set to **Database**).

Since the context's dataset can be used as a data source, one of its practical applications would be for data caching or to support offline data entry. For example, assume that we know that department list in our company does not change often so we can safely cache it (in the case of offline data entry capability, it is a necessity to have a local cache of the department list).

1. The first time, the application connects the context to the server (online).
2. The application retrieves the list of departments off the server.
3. It adds the retrieved department data objects to the context's dataset.
4. It disconnect the context (offline).
5. It now sets the **DataSource** property of the context's Options to **Dataset**.
6. It then serializes the context and store the resulting string in the computer's local harddrive.

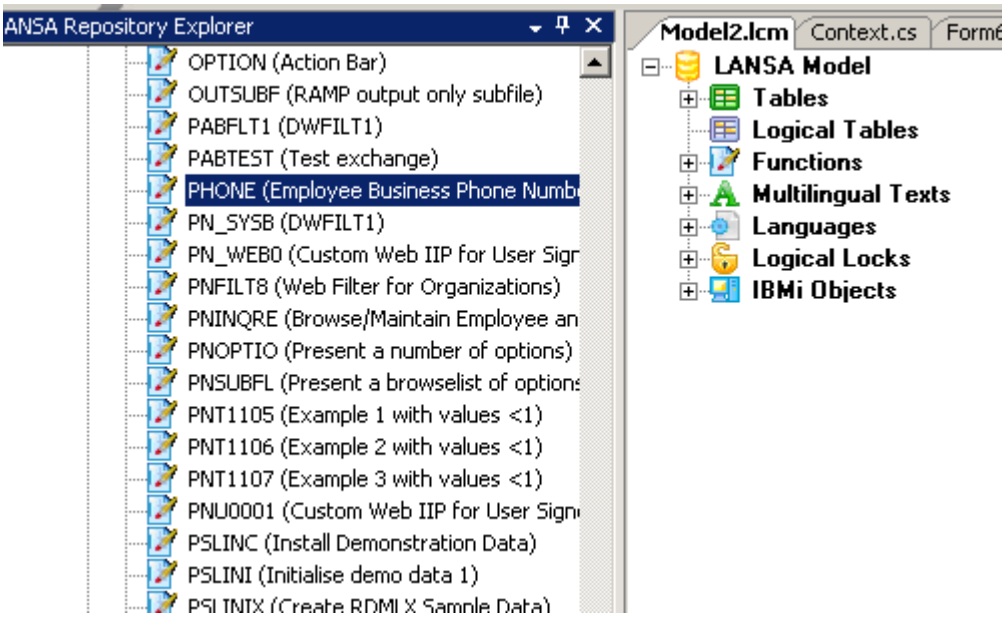
```
// First time, retrieve the department list off the server
context.Connect("username", "password");
Department[] departments = context.Departments.RetrieveList();
context.Disconnect();

// Add the deparments to the context's dataset
context.DataSet.Add(departments);

// Serialize the context
context.Options.DataSource = DataSourceType.Dataset;
string serialized = context.Serialize();

// Save the serialized string somewhere in the local computer.
System.IO.File.WriteAllText("path", serialized);
```

7. The next time the application is run, the context will be deserialized – the dataset will contain the department list.
8. When the application calls the **RetrieveList** method of the **Department** table, the existing departments in the context's dataset will be returned.

```
// Deserialize the context
// Get the serialized string stored in the local computer
string serialized = System.IO.File.ReadAllText("path");
DataContext context = DataContext.CreateFromSerialized(serialized);

// Context remains offline.
// Before we serialized the context we have set the 'DataSource'
// property to 'Dataset' (the serialization process remembers that setting).
// The RetrieveList below will retrieve the existing department objects
// stored in the dataset.
Department[] departments = context.Departments.RetrieveList();
```

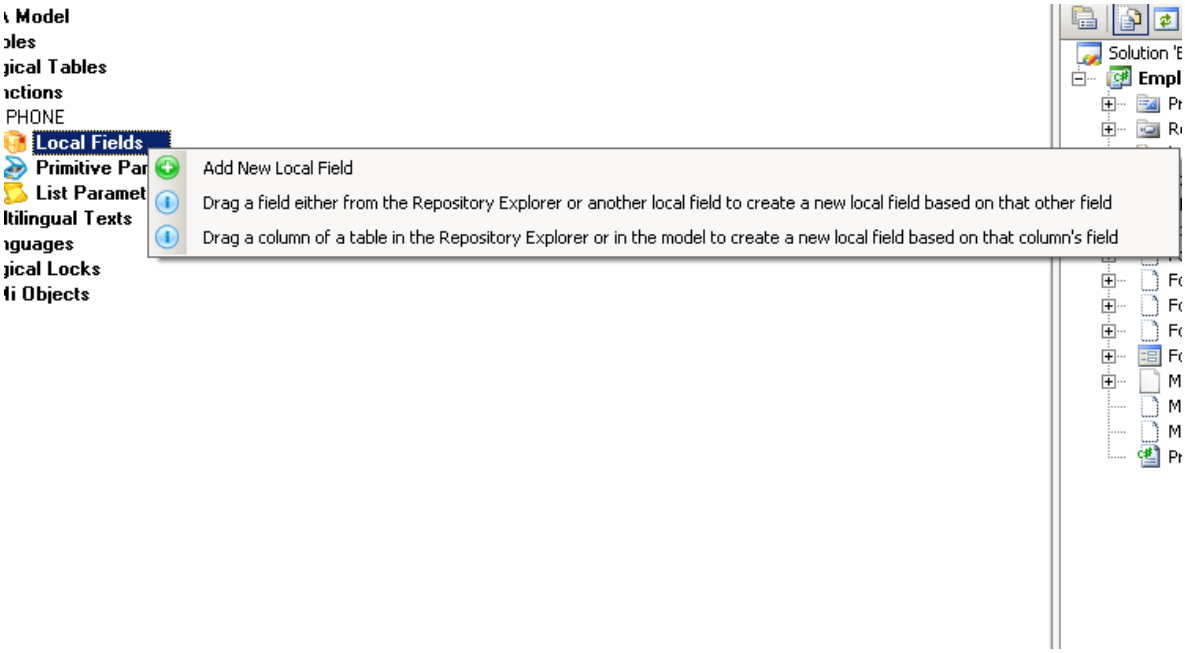
## Creating a Server Function Definition in the Data Model

- We start by dragging a function from the LANSA Repository Explorer and dropping it on our data model document.



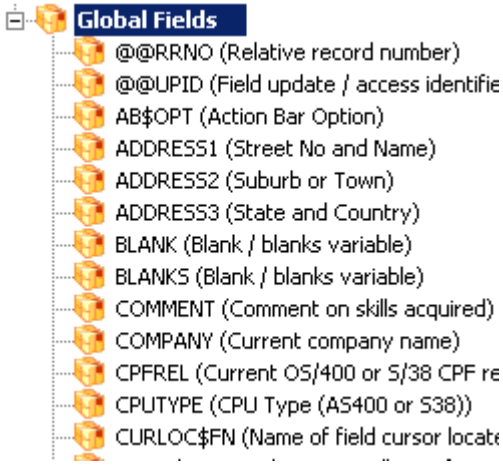

- **Local Fields** are fields used by the function that are defined locally (in the function).
  **Primitive parameters** are exchanged fields.
  **List Parameters** are working lists.


- If the function uses fields that are defined locally in the function (not in the LANSA Repository), we should define those fields in our model.



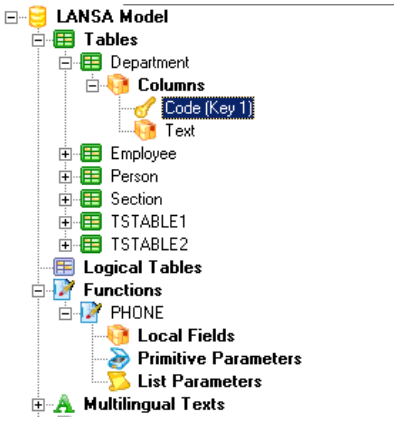

If we do a mouse right-click on the **Local Fields** node, we can see that there are 2 ways to define a local field:

1. Define a completely new field.
2. Define a new field that is based on another field (reference field). There are 3 ways we can do this:
    a. Drag a global field from the LANSA Repository Explorer and drop on the ***Local Fields*** node.

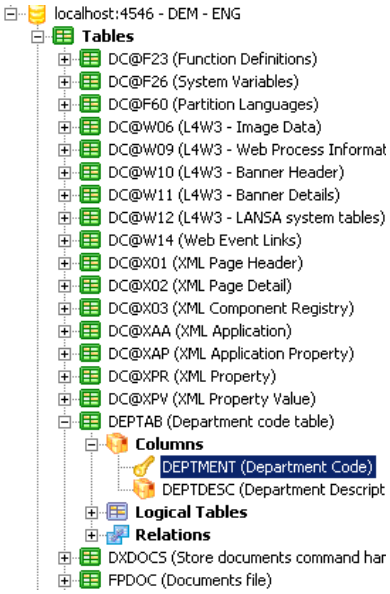



b. Drag a column of a table from the LANSA Repository Explorer or from a table in our model. When we drop the column on the ***Local Fields*** node, the field that the column is based on will be used as the base field for the new field.

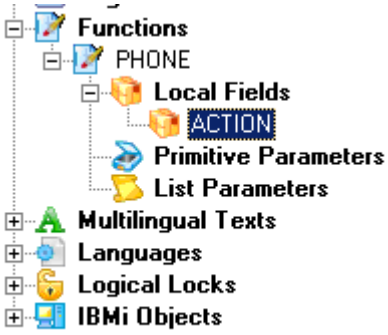
***Data Model***




***Repository Explorer***

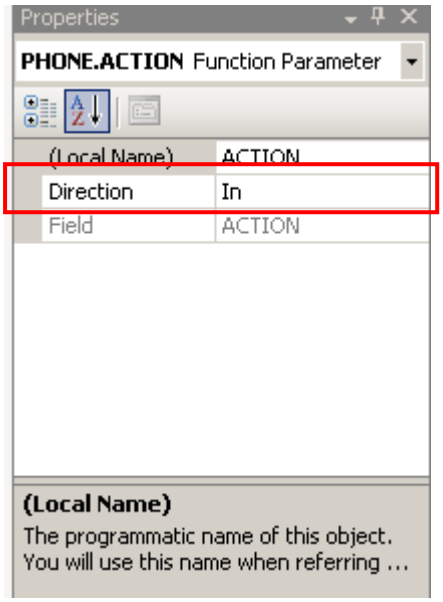



c. Drag another local field to create a new field based on that local field and drop on the ***Local Fields*** node.




- To add a primitive parameters, there are three ways of doing that:
  - Drag one of the fields under the ***Local Fields*** node and drop it on the ***Primitive Parameters*** node.
  - Drag a global field from the LANSA Repository Explorer and drop it on the ***Primitive Parameters*** node.
  - Drag a column of a table in the LANSA Repository Explorer or a column of a table in our model document.

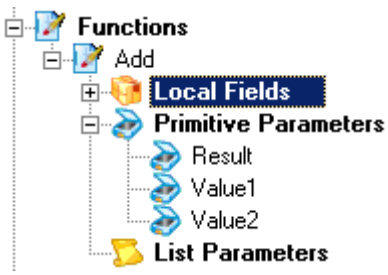
  We must indicate if a parameter is an input or output parameter.




- Adding a list parameter should be straightforward. The list's columns are the fields contained within the working list. List's columns can be added in the same way as primitive parameters.

## Invoking a Server Function

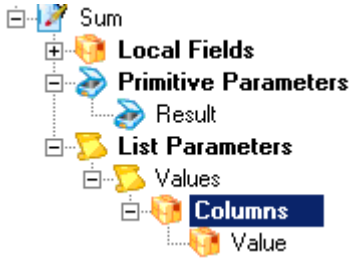
Assuming that we have a function called ADD on the server and we have created its definition on our data model.



**Value1** and **Value2** are input parameters, **Result** is an output parameter.

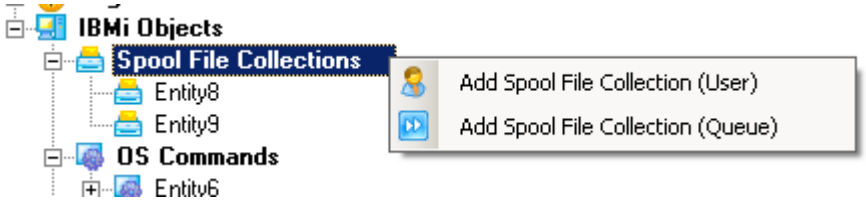
```
// Create an 'invoke info' object.
// An 'invoke info' object is where you put all the parameters
// you need to pass when calling the server function.
// After the function is executed, any values returned by the server
// are available in the 'invoke info' object as well.
AddInvokeInfo invokeInfo = context.Add.CreateInvokeInfo();
invokeInfo.In.Value1 = 5;
invokeInfo.In.Value2 = 10;

// Invoke the function, passing the parameters
context.Add.Invoke(invokeInfo);

// Get the result returned by the function
long result = invokeInfo.Out.Result;
```

## Invoking a Server Function with List Parameters




```
// Create an 'invoke info' object.
// An 'invoke info' object is where you put all the parameters
// you need to pass when calling the server function.
// After the function is executed, any values returned by the server
// are available in the 'invoke info' object as well.
SumInvokeInfo invokeInfo = context.Sum.CreateInvokeInfo();

// Add values to sum to the list
Sum_ValuesItem item1 = new Sum_ValuesItem();
item1.Value = 10;
invokeInfo.Lists.Values.Add(item1);

Sum_ValuesItem item2 = new Sum_ValuesItem();
item2.Value = 10;
invokeInfo.Lists.Values.Add(item2);

// Invoke the function
context.Sum.Invoke(invokeInfo);

// Get the result
long result = invokeInfo.Out.Result;
```

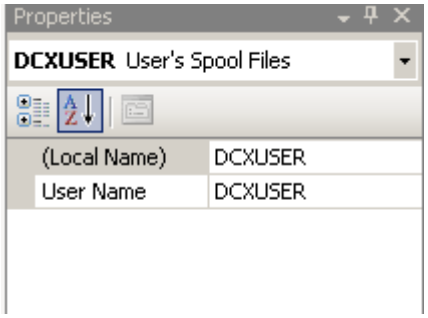
## Using IBM i Spool File Collections




A ***spool file collection*** represents a list of spool files on an IBM i system. There are 2 types of spool file collections:

- Spool file collections for users
- Spool file collections for queues

As an example, let's create a ***spool file collection*** for a user called **DCXUSER**.




After we save the model, we can access the spool file collection called **DCXUSER** through `context.SpoolFileCollections`.

To iterate through the spool files in this collection:

```
foreach (SpoolFile spoolFile in context.SpoolFileCollections.DCXUSER)
{
        Console.WriteLine(spoolFile.JobName);
}
```

We can also reference the spool files by index.

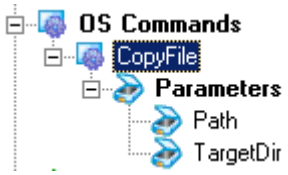
```
SpoolFile firstSpoolFile = context.SpoolFileCollections.DCXUSER[0];

// Get the content of the spool file
String content = firstSpoolFile.Content;
```

## Using IBM i Operating System Command

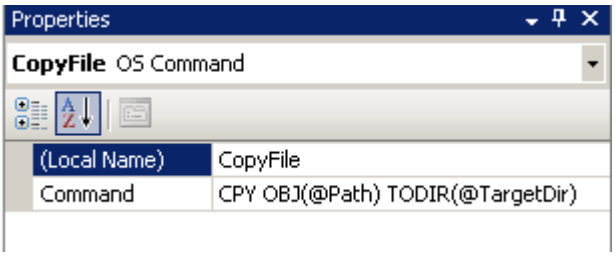
Let's define a command called ***CopyFile*** that has 2 parameters:

- Path
- TargetDir

This command copies a file as specified in the **Path** parameter to the directory specified in **TargetDir**.



This is how the command looks like:




Note that parameters are indicated by prefixing the parameter name with the '@' character.

To execute the command:

```
context.OSCommands.CopyFile.Execute("/test/file1.txt", "/test2");
```