Magic xpa .NET Tutorial

OUTPERFORM THE FUTURE™

This document describes a collection of samples that give you hands-on experience with the .NET capabilities.

This document does not provide full descriptions of each feature added. For more information about each feature, refer to the Magic xpa Help.

This package includes a full working sample of using the .NET capabilities. By following this guide, we will create a project from scratch. You can refer to the samples project in case you encounter any problems.

Defining the .NET Assemblies

To start working with .NET, you need to define the assemblies that you will use.

The assembly definition is done in the Composite Resource Repository (CRR).

1. Open the CRR (Shift+F7) and create a new line.
2. Set the **Type** as **.NET**.
3. Return to the **Name** field and zoom (F5) to open the .NET Assembly Selection list.
4. You can now select your assembly from a list of installed assembly or from any other location. Select the **System** assembly.
5. Create a new line and select the **System.Drawing** assembly.
6. Create a new line and select the **System.Windows.Forms** assembly.

That's it; the assemblies are now ready for use.

Using .NET Controls

To use .NET controls, you need to define the .NET control Object Type.

The Object Type can be defined in either the .NET control model or a .NET variable.

Defining .a .NET Control Model

We will now define a .NET model to be used in a control on the form.

Note that you can use .NET controls without defining a model, as described later on.

1. Create a model with **GUI Display** or **Rich Client Display Class** and set its **Attribute** to **.NET**.
2. Open the property sheet of the model and define an **Object Type**, such as **System.Windows.Forms.MonthCalendar**.

   Note that you can use the auto-complete feature by pressing **Ctrl+Space** and whenever you insert a dot, a selection list with the available options will be opened (just like when you select a Magic xpa internal function).
3. Create a new program.
4. Go to the form of the program and put a .NET control on the form.
5. Assign the new model to the .NET control.

   Note that only models whose object type inherits from **System.Windows.Forms.Control** can be used for .NET controls on a form.

You've finished defining a .NET control. You can now run the new program and see the .NET calendar control on your program.

Note: For Rich Client programs, you also need to define a variable of any type.

Program #3 for Online and #33 for Rich Client in the **samples** project are finished versions of this program.

Changing the Control Properties

We will now change the .NET properties of the .NET control.

1. Open the program that you created, zoom into the form and select the .NET control.
2. Open the property sheet of the control and go to the .NET section.
3. Zoom into the **Object Properties** property.
4. You can now change the properties of the .NET control just as you would have done in Visual Studio. For example, change the **FirstDayOfWeek** property to **Tuesday**.

That's it; you can run this program and see that the left column in the calendar is Tuesday.

Program #4 for Online and #34 for Rich Client in the **samples** project are finished versions of this program.

Notes:

- The simple properties can be changed by entering the value directly or via a combo box.
- The complex properties that expect a .NET object as their value can only be set via an expression, as explained later on.
- You can define the simple properties in the model level.

Binding Data to a .NET Control

You can easily bind data to the control using the control's **Data** property, exactly as you use Magic xpa built-in controls.

Since .NET controls have many properties and many events, before attaching a variable to the **Data** property, you will need to define which .NET property will get the variable value and which .NET event is the trigger for setting the variable with the .NET property value.

This definition is done in the model level in the **Value Property Name** and **Value Changed Event Name** properties.

You will now add the .NET binding declarations to your .NET control model.

This definition can be done once per each control type for which you want to bind data.

1. Open the property sheet of the .NET control model that you created.
2. Set the **Value Property Name** property to the property that will be bound to the data variable, such as **SelectionStart**.
3. Set the **Value Changed Event Name** property to the event that will be the trigger for updating back the data variable, such as **DateChanged**.

Your model can now be used to bind a variable data to a .NET control.

You will now bind a variable to the .NET control.

4. Open the program that you created.
5. Create a variable in this program and set its **Attribute** to **Date**.
6. Zoom into the form and select the .NET control.
7. Open the property sheet of the control.
8. Assign the variable you created to the **Data** property.

You've finished binding the Date variable to the Month Calendar .NET control. To see that the value is bounded, we will now update the Date variable and put it on the form.

9. Go to the Date variable that you created and set an **Init** expression, such as **Date()**.
10. Zoom into the form and add an expression with your variable. You cannot put the variable directly on the form since it is already attached to the .NET control.

That's it; you can now run the new program and see that the .NET calendar control is updated according to the Date variable and that any change in the .NET calendar control value is updated in the Date variable.

Program #5 for Online and #35 for Rich Client in the **samples** project are finished versions of this program.

Defining a .NET Variable

When using .NET objects, you can perform additional actions on the .NET object other than getting the mapped property value, such as get or set other properties, execute methods and respond to events.

We will now define a .NET variable to be used for these actions in our control.

Note that you can execute generic .NET methods and handle generic .NET events even without defining a variable, as described later on.

1. Create a new program.
2. Create a variable in this program and set its **Attribute** to **.NET**.
3. Open the property sheet of the variable and define an **Object Type**, such as **System.Windows.Forms.MonthCalendar**.

   Note that you can use the auto-complete feature by pressing **Ctrl+Space** and whenever you insert a dot, a selection list with the available options will be opened (just like you select a Magic xpa internal function).
4. Go to the form of the program and put this variable on the form or attach it to the **.NET Object** property of a .NET control.

   Note that only variables whose object type inherits from **System.Windows.Forms.Control** can be used as a control on a form.

You've finished defining a .NET variable. You can now run the new program and see the .NET calendar control on your program.

Program #6 for Online and #36 for Rich Client in the **samples** project are finished versions of this program.

Handling .NET Events

We will now react to the events raised by the .NET object.

1. Open the program and go to the **Logic** tab.
2. Create a new event handler with the **Event Type** set to **.NET**.
3. Select the **Calendar** variable as the variable.
4. Select the **DateSelected** event.
5. Add a Verify operation with **'Hello World'** as the message text.

You can now run this program and see that when clicking on a date, the message is displayed.

Note: Events raised by .NET objects that are not placed on the form are handled differently in Rich Client tasks than Magic xpa events. Refer to the *Magic xpa Help* for additional information.

Getting the Value of a Property

We will now get the value selected in the .NET object.

To get the value of a .NET object property, you can simply write the .NET variable, followed by a dot (.) and then the property.

Note that when you press the dot, auto-complete is activated. You can reopen the selection list by pressing **Ctrl+Space** (just like you select a Magic xpa internal function).

We will use the **SelectionStart** property of the variable to get the selected date.

Note that the **SelectionStart** property is a .NET **DateTime** property and it has to be converted into a string in order to show it in a Verify operation.

We can do that by simply using the Magic xpa **DStr()** function.

- In the event handler defined earlier, replace the **'Hello World'** text with the following expression: **DStr(A.SelectionStart,'DD/MM/YYYY')**

Magic xpa automatically converts the .NET type into a Magic xpa type, so it can be used directly in any of the Magic xpa functions.

That's it; you can run this program and see that upon clicking on a date, the correct date is displayed.

Invoking a .NET Method

Instead of using the Magic xpa **Dstr()** function to convert the **DateTime** value into a short string, we could have used the **ToShortDateString()** .NET method.

We can run this method directly on the value we got from the **SelectionStart** property. There is no need to save the property value into a temporary variable in order to run .NET methods on it.

- In the event handler defined earlier, replace the expression: **DStr(A.SelectionStart,'DD/MM/YYYY')** with **A.SelectionStart.ToShortDateString()**

That's it; you can run this program and see that upon clicking on a date, the correct date is displayed.

Program #7 for Online and #37 for Rich Client in the **samples** project are finished versions of this program.

Updating the Value of a .NET Property

To dynamically change the value of a .NET property, you should use the **DNSet()** function.

This function allows you to update any .NET property with the proper value. For example:

1. Create a new event handler, say a System event raised by **Ctrl+1**, and add an Evaluate operation with the following expression:

   DNSet(A.ShowWeekNumbers,'TRUE'LOG)

   This will show the week numbers at the beginning of each line.

2. Run the program and press **Ctrl+1**. Notice that the calendar display changed accordingly.

This update is simple since we use a simple value to update the property. You can also update a property with an enumerator value (as we will see in the next sample) or with a result of another method (as we will see later on).

Program #8 for Online and #38 for Rich Client in the **samples** project are finished versions of this program.

DotNet Internal Code

There are lots of times in which you need to access .NET in order to select a class or some values. For example, if you want to set the Background color of the calendar control to blue, you can use the **System.Drawing.Color.Blue** enumerator.

To directly access .NET objects, you can use an internal code called **DotNet** as a prefix of the required .NET object. For example, **DotNet.System.Drawing.Color.Blue** will return the required value.

So, if you want to change the background color of the calendar control, you can simply:

1. Create a new event handler, say a system event raised by **Ctrl+2,** and add an Evaluate operation with the following expression:

   DNSet(A.BackColor,DotNet.System.Drawing.Color.Blue)

2. Run the program and press **Ctrl+2**. Notice that the calendar display changed accordingly.

Note that the appearance of the calendar control will be changed only if the Use Windows XP Themes

environment setting is set to No.

Earlier, you saw that you can run a .NET method directly in the Expression Editor when using a .NET variable. This is also true for the internal **DotNet** keyword.

So, if you wish to change the background color of the calendar control to a specific color, say with RGB value of (250,30,20), you can use the **System.Drawing.Color.FromArgb()** method:

1. Change the Evaluate operation to the following expression:

   DNSet(A.BackColor,DotNet.System.Drawing.Color.FromArgb(250,30,20))

2. Run the program and press Ctrl+2. Notice that the calendar display was changed accordingly.

In the same way, if you wish to change the font of the calendar control, you can use the **System.Drawing.Font()**method:

3. Add an Evaluate operation with the following expression:

   DNSet(A.Font,DotNet.System.Drawing.Font('Times New Roman',10))

Program #8 for Online and #38 for Rich Client in the **samples** project are finished versions of this program.

Handling Exceptions

Magic xpa provides two functions that help you handle .NET exceptions that occurred when using a .NET object:

- **DNExceptionOccurred** – Informs you that an error occurred during the last .NET operation.
- **DNException** – Returns a reference to the last exception that occurred by a .NET operation.

We will perform an illegal operation, such as setting the date to February 30th.

1. Create a new event handler, say a system event raised by **Ctrl+3** and add an Evaluate operation with the following expression:

   A.SetDate(dotnet.System.DateTime(2001,2,30))

2. Add a Verify operation with the following expression: **DNException().Message** and the following expression as the condition: **DNExceptionOccurerd()**
3. Run the program and press Ctrl+3. Notice that the error message is displayed. If you set the date to a proper value, you will see that the message is not shown.

Program #9 for Online and #39 for Rich Client in the **samples** project are finished versions of this program.

Aliases

As you have seen, there were a lot of times that we used the 'System.Windows.Forms' and 'System.Drawing' prefixes. To make it easier, Magic xpa allows you to create shortcuts to .NET objects by defining .NET aliases.

1. Open the CRR (Shift+F7) and click the **.NET Aliases** button.
2. Add a new line and define your shortcut. For example, define the name of 'Forms' as 'System.Windows.Forms'.

That's all there is to it; from now on, using the word **Forms** is equal to writing **System.Windows.Forms**.

You can define an alias for anything you like, so for example, you can define an alias for **System.Windows.Forms.MonthCalendar** and use it as the Object Type of the .NET variable in the initial example.

Initializing a .NET Variable

Up until now, we used a .NET variable that was associated to a control. This variable will be initiated automatically when the form is created. However, you will need to use .NET variables that are not associated to controls. Those variables should be initialized by a constructor.

In the following example, we will add an icon to the system tray area in the Windows task bar.

Create a new program:

1. Create a variable in this program and set its **Attribute** to be **.NET**.
2. Open the property sheet of the variable and define an **Object Type**, such as **Forms.NotifyIcon**. Note that **Forms** is the alias added previously. Of course you can use the full name and define the object type as: **System.Windows.Forms.NotifyIcon**

Since this variable does not exist as a control on the form, you will need to initialize it manually. You can do that by updating it with an existing .NET object or by running a .NET method that creates a .NET object.

3. Create a handler for the Record Prefix and in it add an Update operation to update your variable with **DotNet.Forms.NotifyIcon()**

The variable is now pointing to an initialized .NET object.

This object will be disposed when the Runtime engine terminates. Since we will run this sample several times from the Studio, the Runtime engine is not terminated, so we will need to dispose the .NET object manually.

4. Create a handler for the Task Suffix and, in it, add an Evaluate operation. Set the expression to: **A.Dispose()** and set the condition to: **Not IsNull(A)**

That's it; the NotifyIcon .NET object will now be disposed when the program is closed.

Now, we will update the .NET object with the icon and text:

1. Create a new line in the Record Prefix logic unit with an Evaluate operation. Set the expression to:
   DNSet(A.Icon,DotNet.System.Drawing.Icon(ServerFileToClient(Translate('%WorkingDir%Images\Mgxpa.ico'))))

   The **ServerFileToClient(Translate('%WorkingDir%Images\Mgxpa.ico'))** will copy the **Mgxpa.ico** file from the **%WorkingDir%Images** folder on the server to the client, so that this image can be used in the .NET object.
2. Create a new line in the Record Prefix logic unit with an Evaluate operation. Set the expression to: **DNSet(A.Text,'Click me')**
3. Create a new line in the Record Prefix logic unit with an Evaluate operation. Set the expression to: **DNSet(A.Visible,'TRUE'LOG)**

You can now run the program and see that a new icon was added to the system tray area.

Program #11 for Online and #41 for Rich Client in the **samples** project are finished versions of this program.

We will now add some logic to the program, such as reacting to a click on the icon. This is similar to the previous sample.

1. Open the program and go to the **Logic** tab.
2. Create a new event handler with E**vent Type** set to **.NET**.
3. Select the .NET variable as the object.
4. Select the **DoubleClick** event.
5. Add a Verify operation with **'Hello World'** as the message text.

That's it; you can run this program and see that when double clicking on the icon, the message is displayed.

Program #12 for Online and #42 for Rich Client in the **samples** project are finished versions of this program.

Enhancing the Sample

We will now enhance the sample to emulate an incoming message alert.

This can easily be done using the .NET **ShowBalloonTip** method.

1. Open the program and go to the **Logic** tab.
2. Create a new event handler with the **Event Type** set to **Timer** and define a **5 second** interval. You can enhance this Timer event to check for changes in your application.
3. In the event handler, add an Evaluate operation with the expression: **DNSet(A.BalloonTipText,'Message arrived')**
4. Add another Evaluate operation with the expression: **A.ShowBalloonTip(30)**

You can now run the program and see that a balloon appears every 5 seconds above the icon.

When selecting the **ShowBalloonTip** method, you may have noticed that this method has two available syntaxes. When this is the case, you can browse between them using the arrows in the tooltip.

Note that when the tooltip is reopened, it will try to find the matching syntax. If there are duplicated syntaxes that match the method arguments, the first method syntax will be shown.

We will now use a different overloaded method:

1. In the event handler, delete the Evaluate operation with the expression:

   DNSet(A.BalloonTipText,'Message arrived')
2. Change the second Evaluate operation expression from:

   A.ShowBalloonTip(30) to

   A.ShowBalloonTip(30,'Information','Message arrived',DotNet.Forms.ToolTipIcon.Info)

That's it; run the program and see that a balloon appears every 5 seconds above the icon, and it also has more features, such as an icon and header text.

Program #13 for Online and #43 for Rich Client in the **samples** project are finished versions of this program.

Further Enhancing the Sample

We will now enhance the sample in further and add a context menu to the notify icon.

This can be easily done by associating an object of class **ContextMenuStrip** to the notify icon.

1. Open the program and go to the **Data View** tab.
2. Create a variable in this program and set its **Attribute** to**.NET**.
3. Open the property sheet of the variable and define an **Object Type**, such as **Forms.ContextMenuStrip**

Since this variable does not exist as a control on the form, you will need to initialize it manually as we did for the **NotifyIcon** variable.

4. In the Record Prefix logic unit add an Update operation to update your variable with **DotNet.Forms.ContextMenuStrip()**

We will now add items to the context menu. To do so, we will use the **Items.Add()** method of the **ContextMenuStrip** variable.

5. In the Record Prefix **handler** add two Evaluate operation with the expressions: **B.Items.Add('Open Message')** and **B.Items.Add('Exit')**

We will now attach the **ContextMenuStrip** object to our **NotifyIcon** variable. We will do this by setting the **ContextMenuStrip** property of the **NotifyIcon** variable with the new **ContextMenuStrip** variable that you just created.

6. In the Record Prefix logic unit add an Evaluate operation with the expression: **DNSet(A.ContextMenuStrip,B)**

You can now run the program and see that a context menu appears after right clicking on the icon.

We will now add an event handler to handle clicks on the context menu items.

1. Open the program and go to the **Logic** tab.
2. Create a new event handler with the **Event Type** set to **.NET**.
3. Select the **ContextMenuStrip** variable as the object.
4. Select the **ItemClicked** event.
5. Add a Verify operation with **'Show Message'** as the message text with a condition of **F.ClickedItem.Name='Open Message'**.
6. Add a Raise Event operation with the internal **Exit** event with a condition of **F.ClickedItem.Name='Exit'**.

That's it; run the program and see that no form is seen and the icon context menu behaves as it should. Click **Exit** in the context menu to close the program.

Program #14 for Online and #44 for Rich Client in the **samples** project are finished versions of this program.

Note that instead of using one generic event and having conditions in it, you could define a .NET variable of type **System.Windows.Forms.ToolStripMenuItem** for each of the menu item entries, update these variables with the result of the **B.Items.Add()** method and then define a separate Event logic unit for the **Click .NET** event on each one of these variables, so that each logic unit will handle the operations for a single entry.

Casting and Reference

Most of the time, when using Magic xpa values with .NET (such as updating a .NET object property with a Magic xpa value or sending a Magic xpa value to a .NET method), there is an automatic conversion between the Magic xpa type and the .NET object type (and vice versa).

However, there are cases in which the conversion is not unique (for example, the method is overloaded and can receive both Int and Long variables) or that you need to specifically convert the value to a .NET type.

For these cases, you can use the DNCast() function to specifically do the casting between the Magic xpa type to the desired .NET type.

For example, let's look at the **System.Text.StringBuilder** class. This class has a method called **Append**, which accepts a string and appends it to the content of the Object instance. However, the **Append** method is overloaded and can receive different classes, including: String, Char, and Char[]. When using the method as-is, by evaluating the **A.Append('abc')** expression, Magic xpa cannot know to which one of the classes the **'abc'** string should be converted. In this case, the Checker will return an error for this expression.

To resolve this conflict, simply use **DNCast('abc',DotNet.System.String)** to specifically cast the **'abc'** string to a **System.String** class, so that the full expression will be **A.Append(DNCast('abc',DotNet.System.String))**.

Program #16 for Online and #46 for Rich Client in the **samples** project are finished versions of this program.

Sometimes you will have to send .NET variables as a reference into a .NET method.

You can do that by using the **DNRef()** function.

We will see some usages of sending variables 'by ref' later on.

Executing .NET Code

Magic xpa provides you with the ability to directly execute .NET code from within your program.

This can be done via a new option added to the Invoke operation.

We will start with a simple program that sums up two numeric values.

1. Create a new program.
2. Create three numeric variables in this program (for the two numbers and the result).

3. Create a User event named **Calculate** and set the **Force Exit** property to **Editing**.
4. Add a button to the form that will raise this User event.
5. Add a handler for the user event you created.
6. In the handler, add an Invoke .NET operation. Zoom into the **.NET code** property and do the following:
    a. Set the method name to **Sum**. This is the name of the .NET method that will be created.
    b. Zoom into the **Arguments** field and:
        - Create a new line and select the first numeric variable.
        - Notice that by default, the **.NET Type** field was updated with a recommended .NET type to match the Magic xpa type of the variable. You can change this value if you need to.
        - Go to the **.NET Variable** column and enter a name for the variable, such as **num1**. This name will be used in your .NET code.
        - Do the same for the second variable.
        - Close the Arguments screen.
    c. Zoom into the **Return Value** field and select the third variable (the result). Notice that by default, the .NET **Return Value Type** field was updated with a recommended .NET type to match the Magic xpa type of the variable. You can change this value if you need to.
    d. Notice that the **.NET Code** section was changed accordingly.
    e. Now you can write your code. Type the following line in the body of the inner method: **return num1 + num2;**
    f. Close the **.NET Code** screen.

That's it; run the program, insert some values into the variables and click the button. See that the result variable was updated accordingly.

Program #18 for Online and #48 for Rich Client in the **samples** project are finished versions of this program.

Executing Complex .NET Code

We have seen simple execution of .NET code to perform a specific operation and return a value. However, you can use this feature to execute complex .NET code, such as creating classes and using them in that code.

Note that you can select and copy the entire code, so it will be easier for you to edit it and verify it with external tools.

In the next example, we will use an Invoke .NET code to:

1. Define a new form class.
2. Open a new form based on the new class.
3. Raise an event from the .NET form and handle it in Magic xpa.
4. Update a value in Magic xpa field when there is a change in a .NET field and vice versa.

5. Manipulate the .NET form from Magic xpa.
6. Close the .NET form from Magic xpa.

Run program #19 for Online and #49 for Rich Client in the **samples** project.

1. In Magic xpa, click the **Open form** button. See that a new semi-transparent, round .NET form opens. You can drag it anywhere on your desktop by pressing and holding the left mouse button on the form and moving the mouse.
2. In the round form, click on the button. This click is caught in the Magic xpa program and as a result, a text on the Magic xpa form will become visible and hidden alternately.
3. In the round .NET form, enter some text into the Text Box control. See that this text is immediately updated in the Edit control on the Magic xpa form.
4. In the Magic xpa form, enter some text into the Edit control. See that this text is immediately updated in the Text Box control on the rounded .NET form.
5. In the Magic xpa form, move the Slider control. See that the opacity of the .NET form changes accordingly.
6. In Magic xpa, click the **Close form** button. See that the round .NET form is closed.

The basic idea behind the program is to use.NET code to handle all the GUI of the .NET round form, and to use regular Magic xpa events to handle the logic.

Of course, this could be done in an external assembly instead of an Invoke .NET operation.

Let's explain how it works:

- For each of the variables on the round .NET form, there is a .NET variable defined in the program. This variable is passed as an argument to the .NET code with indication that only the reference to the variable is sent.
- The matching .NET variable (created for this argument) is updated with the object created on the .NET form, so now the Magic xpa variable has a reference to a .NET object that exists on the .NET form.
- From now on, we can do anything with this object, such as handle an event called **Click** to catch the .NET button click, handle an event called **TextChange** to update a Magic xpa variable any time the .NET variable is changed, or update the .NET variable when there is a change in a Magic xpa variable.
- The .NET code also returns the reference to the .NET form into a Magic xpa .NET variable. This allows us to control the form from Magic xpa, change its properties, such as the **Opacity**, and call its methods, such as **Close**. Note that we could also return the form reference as an argument, in the exact same way as we did for the controls.

Using .NET Choice Controls

You can bind data to the control using the control's **Data** property, exactly as you use Magic xpa built-in controls. (See Binding Data to a .NET Control)

However, as for the internal choice controls, the .NET choice controls also expect to get the items list (also known as the data source object).

You can either do this using .NET commands through the .NET object reference variable or by setting the Magic xpa properties as you do for the internal choice controls.

This section is about using the Magic xpa properties to define the data source of the .NET control.

Since .NET controls have many properties, you will need to define which .NET property will get the data source and which properties will hold the value and display member names.

This definition is done at the model level in the **DataSource Property Name**, **Display Member Property Name** and **Value Member Changed Event Name** properties.

You will now add the .NET data source declarations to your .NET control model.

This definition can be done once per each control type for which you want to set data.

1. Create a model with **GUI Display** or **Rich Client Display Class** and set its **Attribute** to **.NET**.
2. Open the property sheet of the model and define an **Object Type**, such as System.Windows.Forms.ComboBox.

   Note that you can use the auto-complete feature by pressing Ctrl+Space and whenever you insert a dot, a selection list with the available options will be opened (just like when you select a Magic xpa internal function).
3. Set the **Value Property Name** property to the property that will be bound to the data variable, such as SelectedValue.
4. Set the **Value Changed Event Name** property to the event that will be the trigger for updating back the data variable, such as SelectedValueChanged.
5. Set the **DataSource Property Name** property to the property that will receive the item list, such as DataSource.
6. Set the **Display Member Property Name** property to the property that defines the name of the column to be displayed, such as DisplayMember.
7. Set the **Value Member Property Name** property to the property that defines the name of the column that has the data, such as ValueMember.
   That's it, now the model can be used in your programs.

   You will now use the .NET control model in your program.
8. Create a new program.
9. Create a variable in this program. This variable will be used to get/set the control's data.
10. Zoom into the form and put a .NET control on the form.
11. Assign the new model to the .NET control.

    Note that only models whose object type inherits from **System.Windows.Forms.Control** can be used for .NET controls on a form.
12. Assign the variable you created to the **Data** property.
13. Set the choice properties (such as item list and display list) of the control as you do for any internal choice control.

That's it; you can now run the program and see that the item list is seen on the .NET control, that any change in the data variable is seen on the control, and any change in the control is updated to the data variable.

Program #21 for Online and #51 for Rich Client in the **samples** project are finished versions of this program.

Binding the Data View to a .NET Control (Online tasks only)

You can bind the task data view to a .NET control (such as a grid control) and use it instead of the built-in Table control.

Since there can be many .NET controls on the form, you need to specify which .NET control will show and manage the task data view.

This definition is done in the **Dataview Control** property (in the control on the form or in the model level).

Note that this property is enabled if the **DataSource** property name property has a value and the **DisplayMember property name** and **ValueMember property name** properties do not have a value.

When the value of this property is set to **Yes**, you can define the variables that should be seen in the .NET control by zooming into the **Dataview Control Fields** property.

At runtime, a .NET **System.Data.DataTable** object will be automatically created out of the task data view and this object will be attached to the .NET control's property that was defined in the **DataSource property name** property at the model level.

Any change in the Magic xpa variables will be seen on the .NET control and vice versa.

Note: Some grid controls create an additional blank row at the end of the grid. Usually this is governed by a property of the grid control named **AllowUserToAddRows**. This behavior will create inconsistency between the grid and the Magic xpa task since when the user updates this row, Magic xpa is not aware that a new record is being created, so the task will not be changed to Create mode. To avoid this behavior, it is recommended to set this property to **False**. Creation of new records can then be done using the Magic xpa **Create Line** event (F4).

We will now create a program that uses a .NET dataview control to view a data source's content.

1. Create a model with **GUI Display Class** and set its **Attribute** to **.NET**.
2. Open the property sheet of the .NET control model.
3. Set the **Object Type** property to a .NET control that supports **DataSource** objects, such as **System.Windows.Forms.DataGrid**.
4. Set the **DataSource Property Name** property to the property that should get the data, such as **DataSource**.
5. Set the **Dataview Control** property to **Yes**. (This can be done instead in the control properties on the form.)
6. Create a new program.
7. Open the program's properties and set the **Preload View** to **Yes**.
8. Define the program's data view. (Add a data source and some columns.)
9. Go to the form of the program and put a .NET control on the form.
10. Assign the new model as the control's model or assign the .NET variable to the control's **.NET Object** property.
11. Open the property sheet of the control.
12. Zoom into the **Dataview Control Fields** property and define the order of the variables that will be added to the **DataTable** object attached to the .NET control. A value of zero means that the variable will not be available for the .NET control.

That's it; you can now run the program and see that the data view is bound to the **DataGrid** .NET control.

Any navigation between the rows or data update in the .NET control will automatically be reflected in the Magic xpa variables and vice versa.

Program #23 for Online in the samples project is a finished version of this program.

Deployment

When running the application from your clients' machines, if the assembly is not installed on the client, it will be copied to the client machine from the server.

The location on the server side is the location written in the assembly properties in the CRR. You can change this location:

1. Open the CRR (Shift+F7) and park on the assembly line.
2. Park on the **Name** field and zoom (F5) to open the **.NET Assembly** properties dialog box.
3. You can now change the assembly path.

It is recommended to use a logical name for the assembly path, if you need to have different locations for development and deployment.