# GN4 - Code Workflow

*3-august-2015*

## Introduction

The GN4 workflows are XML files containing the definition of activities to be executed in sequential (*SequentialWorkflow*) or interactively (*NavigatorWorkflow*). The xml files are stored into the GN4 database as configuration files.

The old GN4 workflows (*WindowsWorkflow*) are written entirely in *XAML* and based on *WWF* (*Windows Workflow Foundation*); that is, the standard .NET classes (*System.Workflow.Runtime* namespace and others) take charge of loading, parsing and executing the activities.

This approach has some inconvenient, because it is not easy to write *if-else* conditions in XAML or to store values into global variables, etc…

The new **GN4 CodeWorkflow** allows user to write workflows in VB.NET taking advantages of all the .NET features like variable declarations, errors management (*try-catch*), loops (*For Each*) and so on. Moreover, it can be debugged using the testing environment (see below).

The following documentation assumes that the reader is familiar with VB.NET and old GN4 workflows.

## Overview

The new CodeWorkflows are pieces of *VB.NET* code, stored into the database as *CDATA* section of the xml configuration file. Like the old WindowsWorkflows, the CodeWorkflows are always executed on server.

The VB.NET code is read from the database, compiled runtime and cached on server the first time it is executed. The next times it is executed, the compiled workflows are read directly from the server cache.

Note that, from the application’s point of view, the old GN4 WindowsWorkflow and the new GN4 CodeWorkflow are interchangeable. This means that you can write a Code workflow which does the same things than an old WindowsWorkflow and then replace the old <wfRes:workflow> tag and all its content with the new <codeWorkflow> tag’s contents (into, for example, a *workflow* configuration file, into a data source template, etc…)

You can also call a CodeWorkflow from a WindowsWorkflow and vice-versa (using the *ExecuteSequentialWorkflow* activity).

The CodeWorkflow classes are defined into *Server.dll*.

### The structure of the configuration file:

Both the old WindowsWorkflows and the new CodeWorkflows definitions are written into **xml** configuration files having the “**wf\_...**” prefix.

The configuration files are stored into the GN4 database using the “*config*” command of the *cmd4.exe* command-line tool. For example:

*cmd4 config –in ..\config\global\wf\_ImportDocuments.xml*

Let’s compare the structure of the old and new GN4 workflows.

The old sequential GN4 WindowsWorkflow configuration file has a structure like this:

(note that the name of activities and resources, like ‘<Activity1>’, ‘<Resource3>’, etc…, are only to show the workflow structure, not the real ones)

<wfres:workFlow

xmlns:wfres="http://www.teradp.com/schemas/GN4/1/WFRes.xsd">

<SequentialWorkflow>

<Activity1

x:Name="activity1" />

<Activity2

x:Name="activity2" />

<Activity3

x:Name="activity3" />

. . .

</SequentialWorkflow>

<Resource1

wfres:activityName="activity1" />

<Resource3

wfres:activityName="activity3" />

. . .

</wfres:workFlow>

where **<Activity…>** tags are the definition of the GN4 sequential activities (like *LoadObjects*, *TransformXml*, …) and **<Resource…>** are the xml serialized classes (like *XmlExportOptions*, *ObjectUI*, *SearchConditions*, …) needed by the activities to perform their operations. Each resource is associated to the Activity by its *wfres:activityName* property.

Instead, the new sequential GN4 CodeWorkflow is a piece of VB.NET code written into a CDATA section of the xml configuration file. It has a structure like the following:

<codeWorkflow

xmlns="http://www.teradp.com/schemas/GN4/1/WFRes.xsd">

<References>

<!-- Add here references to additional assemblies (DLL) -->

<!-- <Reference>XXXX</Reference>-->

</References>

<Imports>

<!-- Add here additional namespaces to import -->

<!-- <Import>XXXX</Import> -->

</Imports>

<Members>

<![CDATA[

]]>

</Members>

<Sequential>

<![CDATA[

Dim resource1 As Resource1 = New Resource1

Dim resource3 As Resource3 = New Resource3

Dim activity1Act As Activity1 = New Activity1(Context)

activity1Act.Resource1 = resource1

Dim activity1Res As Activity1Result = activity1Act.Do()

Dim activity2Act As Activity2 = New Activity2(Context)

Dim activity2Res As Activity2Result = activity2Act.Do()

Dim activity3Act As Activity3 = New Activity3(Context)

Activity3Act.Resource3 = resource3

Dim activity3Res As Activity3Result = activity3Act.Do()

]]>

</Sequential>

</codeWorkflow>

(note that the name of classes and properties, like ‘Activity1’, ‘Resource3’, etc…, are only to show the workflow structure, not the real ones)

There are many things to note:

* the root of the CodeWorkflow is the tag ***<codeWorkflow>*** (not <workflow> anymore). So, to find the released CodeWorkflows into the daily *GN4Config.zip*, you can looks for the *“<codeWorkflow*” token into the xml files.
* the *VB.NET* code of the workflow is into the *CDATA* section of the ***<Sequential>*** tag
* if the CodeWorkflow contains *VB.NET* Sub()s or Function()s, they must be written into the *CDATA* section of the ***<Members>*** tag.
* The default assemblies used by the CodeWorkflow are: *System.dll, System.Core.dll, System.Xml.dll, System.Xml.Linq.dll, Common.dll* and *Server.dll*.

If the CodeWorkflow needs to use classes defined into assemblies other than the default ones, then you have to add the ***<Reference>*** row into the ***<References>*** tag with the name of the required library.

For example, here the *<References>* of the publishing workflows (they need to use the *gnWebSrv* class in *GnWebClient.dll* and other classes that are not defined into the default assemblies):

*<References>*

*<!-- Add here references to additional assemblies (DLL) -->*

*<Reference>.\GNClient.dll</Reference>*

*<Reference>.\GnWebClient.dll</Reference>*

*<Reference>.\Microsoft.Web.Services3.dll</Reference>*

*<Reference>System.Data.dll</Reference>*

*<Reference>System.Web.dll</Reference>*

*<Reference>System.Web.Services.dll</Reference>*

*</References>*

* if the CodeWorkflow needs to extensively use a .NET namespace, you can import it adding the ***<Import>*** row into the ***<Imports>*** tag (this way, you avoid to write it many times into the code).

For example:

<References>

<!-- Add here references to additional assemblies (DLL) -->

<Reference>GNClient.dll</Reference>

</References>

<Imports>

<!-- Add here additional namespaces to import -->

<Import>TeraDP.GN4.GNClient</Import>

</Imports>

### **Activities**

A workflow is a list of activities to execute. There are several GN4 activities that you can use to perform operations in GN4 system (like reading GN4 objects from database, sending emails, creating new GN4 objects, publishing, etc…).

The CodeWorkflow activities are .NET classes, so you have to instantiate them into the VB.NET code before using them. Let’s see how.

In the following example, we want to load the xml of a GN4 object, given its id.

Dim imgId As Integer = 12345

Dim loadImgAct As LoadObjects = New LoadObjects(Context) With {.Name = "load picture", .Description = "loading image..."}

loadImgAct.ObjectIds.Add(imgId)

Dim loadImgRes As LoadObjectsResult = loadImgAct.Do()

Dim objectXml As XDocument = loadImgRes.XmlOut

#### ***First, we create a new activity object*** (of type LoadObjects, called ‘loadImgAct’):

Dim loadImgAct As LoadObjects = New LoadObjects(Context) With {.Name = "load picture", .Description = "loading image..."}

(note that in the standard CodeWorkflows, the name of the activity objects ends with the “*…Act*” suffix, to easily identify the variable as GN4 activity into the code)

Note that:

* all the CodeWorkflow activities receive the *ExecutionContext* object (called ‘Context’) as parameter. This object contains the values passed to the workflow (Pars, ObjectIds, Data, etc…); it is analogous to the root activity (often called ‘*mainWorkflow*’) of the old WindowsWorkflows.
* every CodeWorkflow activity object has the Name property, but (unlike the old WindowsWorkflows) it is not used anymore to associate the resource xml to the activity, but only in progress reports and error reporting. In this example, *loadImgAct.Name = “load picture”*.
* another property of the CodeWorkflow activity object is Description and it is used in the progress report to display which activity is being executed.

#### ***Then, we set the input data* (in this example, the id of the object to look for):**

loadImgAct.ObjectIds.Add(imgId)

In CodeWorkflows, all the data needed in input by the activity are properties of the activity object. In this example, the *loadImgAct.ObjectIds* property is a list of integers, so we can simply add as number the id of the GN4 object to look for.

#### **Execute the activity**

Dim loadImgRes As LoadObjectsResult = loadImgAct.Do()

To execute a code activity you must call its Do() method. The *Do()* method returns the output data into a *result object*. Note that every activity type as a corresponding …Result class which contains the operation results: the LoadObject activity returns the read xml into a LoadObjectsResult object, the TransformXml activity returns the transformed xml into a TransformXmlResult object, and so on.

#### **Read the operation results:**

Dim objectXml As XDocument = loadImgRes.XmlOut

In general, every code activity receives data in input, performs an operation and returns the output results. The data in input are the properties of the activity object (like *loadImgAct.ObjectIds*); the output results are the properties of the result object (like *loadImgRes.XmlOut*).

(note that in the standard CodeWorkflows, the name of the result object ends with the “*…Res*” suffix, to easily identify the variable as output objects into the code)

All the significant activities of the old *WindowsWorkflows* have been ported to the new *CodeWorkflows*, keeping the same names.

There are some important exceptions:

* the old Variant activity becomes “Variation” (because “variant” is a reserved *VB.NET* keyword).
* the old XAML activities that manage data lists (*DataOp*, *IdsOp*, …) were not ported in code workflows, because VB.NET can manage lists much better.
* the replicator activities (*DataReplicator*, *ObjectsReplicator*, …) were not ported in code workflows, because VB.NET can manage looping much better (*For Each, Do While*, etc…).

Keep in mind that, like the old WindowsWorkflow, also the CodeWorkflows run always on server. This means that *in CodeWorkflows the file paths must refer to the local disk of server* (or to other folders that can be accessed by the GN4 server).

### **The execution context**

Every activity must receive the *Context* object into its constructor.

Dim loadImgAct As LoadObjects = New LoadObjects(**Context**) With {.Name = "load picture"}

The *Context* object contains all the workflow parameters and has some useful properties and methods:

* ExecutionLog property.

It is a xml variable, which is used in some standard workflow to contain xml data needed by GN4 to work. For example, the *importWF* workflows of the *Feed Data Source* in *Back4* receive the data to process in the *Context.ExecutionLog* variable. In custom code workflows, it can be used to store xml data when needed.

* XsltExtensions object.

It allows to use the *GN4 XSLT extension* methods in code workflows, such as objectIdFromString(), trimText(), pathFileName(), etc…

See <http://tech.teradp.com/tech/html/gn4/docs/VSdoc/frlrfTeraDPGN4CommonXsltExtensionsClassTopic_members--.html> for more information about the XSLT extensions of GN4.

* CreateActivityData static method.

It creates a file object (*IActivityData*) from a string and it can be useful to save a string value into a file.

## **Interactive workflows**:

Besides the **batch** *Sequential* code workflow, there are the *Navigator* workflows: they are **interactive** code workflows that show visual controls to the user and wait for input.

The *Navigator* code workflows are supported only in *GN4 version 2.0* or greater.

The structure of the interactive code workflows is the same than the batch workflows, but the tag name changes from *<Sequential>* to *<Navigator>*, this way:

<codeWorkflow

xmlns="http://www.teradp.com/schemas/GN4/1/WFRes.xsd">

<References />

<Imports />

<Members>

<![CDATA[

]]>

</Members>

**<Navigator>**

<![CDATA[

. . .

]]>

**</Navigator>**

</codeWorkflow>

The *<Navigator>* tag tells *GN4* to run the workflow in a visual way; that is, inside a form.

A *Navigator* workflow uses interactive activities (called *interactions*). Here is the list of the main interactions:

* **Upload**: allows user to import external files into GN4 system
* **EditNewObject**: allows user to edit or create a GN4 object
* **EditObjects**: allows user to edit multiple GN4 objects at time
* **Publish**: ask user where publishing a GN4 object and publishes it
* **SearchForm**: searches for GN4 object into GN4 database
* **Crop**: shows the cropping control
* **Multicrop**: allows user to crop multiple pictures at time
* **Preview**: shows the HTML preview of GN4 objects

Here is a sample of the syntax to call the interactive *Upload* activity into a *Navigator* workflow:

Dim uploadAct As Upload = New Upload(Context) With {.Name = "upload files"}

uploadAct.MinFileCount = 1

Dim uploadRes As UploadResult = Await uploadAct.Do()

Dim filePath As String = uploadRes.Data(0).LocalPath

The syntax to call an interactive activity is similar to the sequential one. In fact, just like the sequential activities, we have to:

1. create a new activity object

An interaction is created like a sequential activity, there is no difference. We have to pass the ‘*Context*’ object to the activity and set the activity name. For example:

Dim uploadAct As Upload = New Upload(Context) With {.Name = "upload files"}

1. set the input data

The interaction’s properties are set like the sequential ones, there is no difference.

Every activity has its own input properties. For example:

uploadAct.MinFileCount = 1

1. execute the activity

To execute an interaction we have to call the *Do()* method (just like the sequential activities), but there is an important difference: the Await token.

The syntax to call an interaction inside a *Navigator* workflow is, for example:

Dim uploadRes As UploadResult = Await uploadAct.Do()

The *Await* token suspends the workflow until the user clicks on the *Next* button. It is available only from *.NET Framework 4.5*; this is because the Navigator workflows cannot work in *GN4 1.6*, which uses *.NET Framework 4.0*.

Note that, just like the sequential activities, every type of interaction has its corresponding result class. In this example, the *Upload* interaction saves the operation results into the *UploadResult* object.

1. read the operation results

Into the *Navigator* workflows the output results of the activities are available as properties of the result objects (*UploadResult*, *EditNewObjectResult*, *SearchFormResult*, etc…), just like the sequential activities. For example:

Dim filePath As String = uploadRes.Data(0).LocalPath

### The buttons of the *Navigator* workflow

#### The ‘Next’ and ‘Cancel’ buttons

When an interaction is executed, it shows the visual controls to the user and waits for input. At the bottom of the form are displayed the ‘*Next*’ and ‘*Cancel*’ buttons, like the old *WindowsWorkflow*; so the user can choose if going to the next activity (*Next*) or aborting the execution of the workflow (*Cancel*).

#### The ‘Finish’ button

Because the code workflows are not managed by the *.NET WindowsWorkflowFoundation* engine, GN4 doesn’t know if the current interaction is the last one or not. So, to show the ‘*Finish*’ button instead of the ‘*Next*’ button when the last activity is executed, you have to explicitly set the **ShowFinishButton** property of the last interactive activity of your workflow.

For example:

Dim uploadAct As Upload = New Upload(Context) With {.Name = "upload files", **.ShowFinishButton = True**}

Running this activity, the ‘*Finish*’ button is displayed, to indicate that it is the last interaction of the workflow. However, note that it is only a change of the button label: there isn’t any verification whether the activity is the actual last of the workflow or not.

#### The ‘Back’ button

The ‘*Back*’ is not visible in code workflows by default. This is because, in code workflows, it must be managed by the code workflow itself.

The ‘*Back*’ button is managed using the VB.NET **line-label** and **GoTo** statements.

Here is an example of the syntax used to enable the ‘*Back*’ button into the code workflow:

'---------- first interaction

uploadAct:

Dim uploadAct As Upload = New Upload(Context) With {.Name = "upload"}

. . .

Dim uploadRes As UploadResult = Await uploadAct.Do()

'---------- last interaction interaction

editMetaDataAct:

Dim editMetaDataAct As EditNewObject = New EditNewObject(Context) With {.Name = "editMetaData", .ShowBackButton = True, .ShowFinishButton = True}

. . .

Dim editMetaDataRes As EditNewObjectResult = Await editMetaDataAct.Do()

If Context.GoBack Then GoTo uploadAct

To enable the ‘*Back*’ button in a code interaction:

1. set the **ShowBackButton** property to true.

This will display the ‘*Back*’ button next to the ‘*Cancel*’ and ‘*Next*’ buttons. For example:

Dim editMetaDataAct As EditNewObject = New EditNewObject(Context) With {.Name = "editMetaData", **.ShowBackButton = True**, .ShowFinishButton = True}

1. set a VB.NET ***line-label*** just before the activity target of the ‘*Back*’ button.

In the example, we want that the ‘*Back*’ button of the *EditNewObject* activity returns to the previous *Upload* activity. So, we write the ‘*uploadAct:*’ *line-label* just before the creation of the target *Upload* activity.

**uploadAct:**

Dim uploadAct As Upload = New Upload(Context) With {.Name = "upload"}

1. check the **Context.GoBack** property just after the activity which shows the ‘*Back*’ button.

When the user clicks on the ‘*Back*’ button, the *Context.GoBack* property is set to true. So, we can check this property to determine if retuning to the previous activity or not. For example:

Dim editMetaDataRes As EditNewObjectResult = Await editMetaDataAct.Do()

**If Context.GoBack Then GoTo uploadAct**

## Testing environment:

The cool thing about the *CodeWorkflows* is that you can write the *VB.NET* code using *IntelliSense* and debug it line-by-line using *Visual Studio Express.*

There is a VB.NET solution, called **WorkflowTester**, which we can use to write and debug code workflows, before saving them into the database and running them in a production environment.

***Note that WorkflowTester works only from GN4 2.1.2489.***

#### Setup testing environment

To setup the testing environment for *CodeWorkflows* you have to:

1. copy the **WFTest.zip** archive from the daily build folder to your PC

The zip archive contains:

* the solution ***WorkflowTesterExpress.sln*** to open in *Visual Studio Express* and all the *.vb* source files
* the files where writing the workflow: **SequentialProgram.vb** and **NavigatorProgram.vb**
* (only from GN4 2.2) the source code of the **WFUtils.dll** library (it is a GN4 utility library for workflows)

1. extract all the files on your PC (for example, into a folder called ‘***WFTest***’)
2. copy the GN4 .dlls into the ‘*WFTest\WorkflowTester\bin*’ folder.

The minimum set of GN4 libraries requested by *WorkflowTester* is:

* + GNClient.dll
  + Common.dll
  + GNQuery.dll
  + GNCx.dll
  + WebUI.dll
  + Server.dll
  + Client.dll
  + WinUI.dll
  + WFUtils.dll (only from GN4 2.2)

However, depending on the workflow you are writing, also other libraries could be mandatory; for example, if the workflow calls the *Parse* activity, then you have to copy all the GN4 plugins (*ParseImg*, *ParseMultimedia*, etc…) into the ‘*WFTest\WorkflowTester\bin*’ folder too.

In general, the best option is to decompress all the released archive ([http://tech.teradp.com/tech/download/gnportal/rel21/GN4bin\_64.zip](http://helpdesk.miles33.com/support/scp/l.php?url=http%3A%2F%2Ftech.teradp.com%2Ftech%2Fdownload%2Fgnportal%2Frel21%2FGN4bin_64.zip&auth=a8271f89fdab5b379994fda4ebc29ecb)) into the ‘*WFTest\WorkflowTester\bin*’ folder.

1. configure the connection data into the ‘*WFTest\WorkflowTester\appSettings.xml’* file.

The *appSettings.xml* file is mandatory, because the testing environment needs the data (*database name, username, password, etc…*) to login to the GN4 database.

Note that, if you need to change the path of your *appSettings.xml file*, you have to modify the<appSettings file="..\appSettings.xml"> tag in the *app.config* configuration file of the *WorkflowTesterExpress* *VB.NET* project.

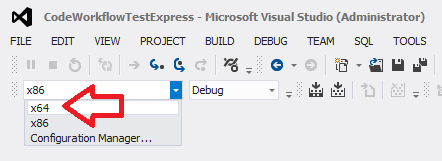
1. open the solution *WorkflowTesterExpress.sln* using *Visual Studio Express.*

***Important: Visual Studio Express must be started as Administrator.***

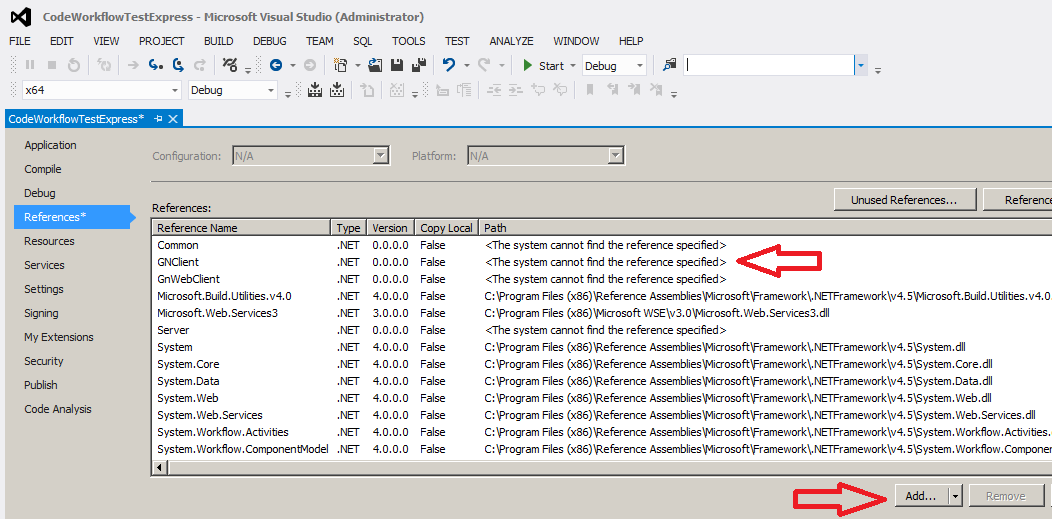
1. try to compile the solution

If something’s wrong in compilation, please check the following settings of *Visual Studio Express*:

* check the building configuration (*x64=64bit, x86=32bit*): it must matches the version of the dlls into the ‘*WFTest\WorkflowTester\bin*’ folder



* check the *References* of the *WorkflowTesterExpress* *VB.NET* project into the solution: into the *SolutionExplorer* window, right-click on the *WorkflowTesterExpress* project, select *Properties* and check the *References* tab). If some referenced library has wrong or missing path, you can try to remove and re-create (Add button) the reference into the project.



#### Writing a code workflow into the testing environment

Once the *WorkflowTesterExpress* *VB.NET* project has been successfully compiled, you are ready to write the code workflow.

The *VB.NET* project contains two files (*SequentialProgram.vb* and *NavigatorProgram.vb*) where you can write your sequential (*batch*) or navigator (*interactive*) workflow.

For example, if you want to write a sequential workflow, double-click on the *SequentialProgram.vb* into the *SolutionExplorer* of *VisualStudio* to open it.

The *SequentialProgram.vb* file is the following:

'<codeWorkflow

' xmlns="http://www.teradp.com/schemas/GN4/1/WFRes.xsd">

Public Class SequentialWorkflow

'--------------------- Sequential workflow sub/functions/fields go here

'<Members>

' <![CDATA[

* write here your Sub and Function methods (if any)

' ]]>

'</Members>

'--------------------- End of sequential workflows sub/functions/fields

Private Sub \_\_Do()

'----------------------- Sequential workflow code goes here

'<Sequential>

' <![CDATA[

* write here your sequential code workflow

' ]]>

'</Sequential>

'----------------------- End of sequential workflow code

End Sub

End Class

'</codeWorkflow>

When running the *WorkflowTester.exe* program and clicking on the ‘*Execute*’ button, the *\_\_Do()* method of the *SequentialWorkflow* class is called.

You have to write your new sequential code workflow into that method, after the “Sequential workflow code goes here <Sequential> <![CDATA[” comments.

On the other hand, if you want to debug an existing code workflow, you have to copy the contents of the “*<Sequential><![CDATA[*“ section of the configuration file (“*wf\_...*” file) into the same place.

Note that *WorkflowTester.exe* can also load automatically existing workflow into the *SequentialProgram.vb* file: see below for details.

For example, to debug the sample workflow seen above in *Activities* paragraph, the *SequentialProgram.vb* file should be:

'<codeWorkflow

' xmlns="http://www.teradp.com/schemas/GN4/1/WFRes.xsd">

Public Class SequentialWorkflow

'--------------------- Sequential workflow sub/functions/fields go here

'<Members>

' <![CDATA[

' ]]>

'</Members>

'--------------------- End of sequential workflows sub/functions/fields

Private Sub \_\_Do()

'----------------------- Sequential workflow code goes here

'<Sequential>

' <![CDATA[

Dim imgId As Integer = 12345

Dim loadImgAct As LoadObjects = New LoadObjects(Context) With {.Name = "load picture", .Description = "loading image..."}

loadImgAct.ObjectIds.Add(imgId)

Dim loadImgRes As LoadObjectsResult = loadImgAct.Do()

Dim objectXml As XDocument = loadImgRes.XmlOut

' ]]>

'</Sequential>

'----------------------- End of sequential workflow code

End Sub

End Class

'</codeWorkflow>

If the code workflow contains *Sub()* or *Function()* *VB.NET* methods, they must be written into the *SequentialProgram.vb* file just after the “Sequential workflow sub/functions/fields go here <Members> <![CDATA[” comments.

For example:

'<codeWorkflow

' xmlns="http://www.teradp.com/schemas/GN4/1/WFRes.xsd">

Public Class SequentialWorkflow

'--------------------- Sequential workflow sub/functions/fields go here

'<Members>

' <![CDATA[

Sub ClearTestObject(ids As IList(Of Integer))

'clear test objects

Dim spikeAct As Spike = New Spike(Context) With {.Name = "spike"}

spikeAct.ObjectIds = ids

Dim spikeRes As SpikeResult = spikeAct.Do()

End Sub

' ]]>

'</Members>

'--------------------- End of sequential workflows sub/functions/fields

Private Sub \_\_Do()

'----------------------- Sequential workflow code goes here

'<Sequential>

' <![CDATA[

Dim imgId As Integer = 12345

Dim loadImgAct As LoadObjects = New LoadObjects(Context) With {.Name = "load picture", .Description = "loading image..."}

loadImgAct.ObjectIds.Add(imgId)

Dim loadImgRes As LoadObjectsResult = loadImgAct.Do()

Dim objectXml As XDocument = loadImgRes.XmlOut

ClearTestObject(New List(Of Integer)({imgId}))

' ]]>

'</Sequential>

'----------------------- End of sequential workflow code

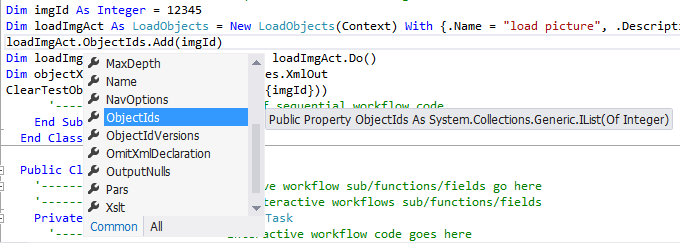
End Sub

End Class

'</codeWorkflow>

If you press **Ctrl+Spacebar** *Visual Studio Express* automatically completes what you are writing.

Moreover, *Visual Studio Express* supports you with IntelliSense while writing a code workflow. This means that if you write, in the above example, “*loadImg.*”, then *Visual Studio Express* shows you all the available methods and properties of the *LoadObjects* activity.

**

#### Running a code workflow into the testing environment

Once you have written your code workflow in *SequentialProgram.vb* file (or *NavigatorProgram.vb* file, if it is an interactive workflow), you can try it running the *WorkflowTesterExpress* project (clicking the **F5** button).

*Visual Studio Express* provides many tools to write and debug the VB.NET code. For example:

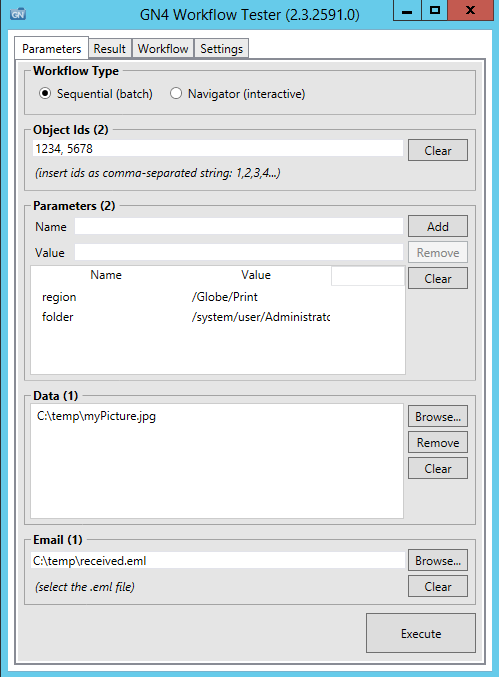
* you can set a break-point to a specific line (**F9**) to block the execution at the line
* when the execution is blocked by a break-point, you can:
  + go to the next line (**F10**)
  + go into a *Sub()* or *Function()* method (**F11**)
  + resume the execution of the program (**F5**)

Check *Microsoft* documentation for more info about the *Visual Studio Express* features.

Creating a breakpoint (**F9**) on the first line of the workflow is a good starting point to debug it.

#### WorkflowTester – Parameters tab

The *WorkflowTester* application looks like this:



Into a real production GN4 environment, the code workflows are called by menu, toolbar, script, etc… receiving input data (name-value parameters, list of GN4 objects ids, list of files, etc…).

In GNPortal there is an *.aspx* test page (<http://localhost/GN4/test/wftestForm.aspx>), where you can specify the input parameters and run a workflow (both *WindowWorkflow* and *CodeWorkflow*) for testing (note that the workflow must already exist into the GN4 database).

Into the ‘***Parameters***’ tab you can set the input parameters of the workflow you are debugging: object ids, name-value parameters and files.

* ‘*Object Ids*’ values are set as a comma-separated list of numbers. They are available in the **Context.ObjectIds** property of the workflow.

Dim loadImgAct As LoadObjects = New LoadObjects(Context) With {.Name = "load picture", .Description = "loading image..."}

loadImgAct.ObjectIds = Context.ObjectIds

Dim loadImgRes As LoadObjectsResult = loadImgAct.Do()

* ‘*Parameters*’ is a list of arbitrary name-value strings. The value of a parameter is available in the **Context.ParValue(‘name’)** property of the workflow.

Dim destPath As String = Context.ParValue("destPath")

If String.IsNullOrEmpty(destPath) Then

Throw New TeraDP.GN4.Common.WorkflowException(TeraDP.GN4.Common.ErrorCode.WFInvalidParam, "destPath")

End If

Dim maxSizeValue As String = Context.ParValue("maxSize")

If String.IsNullOrEmpty(maxSizeValue) Then

Throw New TeraDP.GN4.Common.WorkflowException(TeraDP.GN4.Common.ErrorCode.WFInvalidParam, "maxSize")

End If

Dim maxSize As Integer = CInt(maxSizeValue)

(see below the ‘*Error management’* section for more info about *WorkflowException)*.

* ‘*Data*’ are the full paths of the binary files passed to the workflow. They are available in the **Context.Data** property of the workflow.

Dim callImportDocumentsAct As ExecuteSequentialWorkflow = New ExecuteSequentialWorkflow(Context) \_

With {.Name = "execute workflow", .Description = "Call the documents import workflow"}

callImportDocumentsAct.WorkflowName = "ImportDocuments"

callImportDocumentsAct.Data = Context.Data

Dim callImportDocumentsRes As ExecuteSequentialWorkflowResult = callImportDocumentsAct.Do()

* ‘*Email*’ is the full path of one email file (*.eml*). The *WorkflowTester* program emulates the *Back4* email poller: that is, it reads the email data (saving the body as first attachment) and passes them to the workflow into the **Context.Emails** property.

The *WorkflowTester* program stores the last used values of the fields into a text file called **WorkflowTester.ini**.

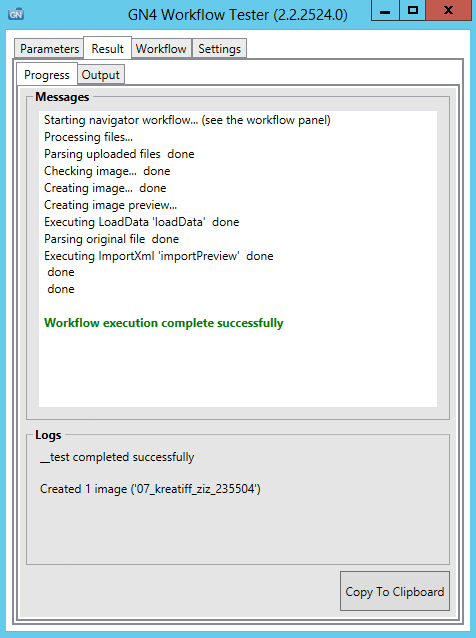
Note that the *Sequential* and *Navigator* workflows have different set of input parameters.

To run and debug the workflow, click on the ‘***Execute***’ button. The button is disabled during the workflow’s execution.

#### WorkflowTester – Result tab

The ***Result*** tab shows the final result of the workflow execution.

The progress messages and the error messages are displayed into the ‘***Result-Progress***’ tab.



The ‘***Result-Output***’ tab shows the messages generated by the *OutputMessage* and *OutputXml* activities (if any).

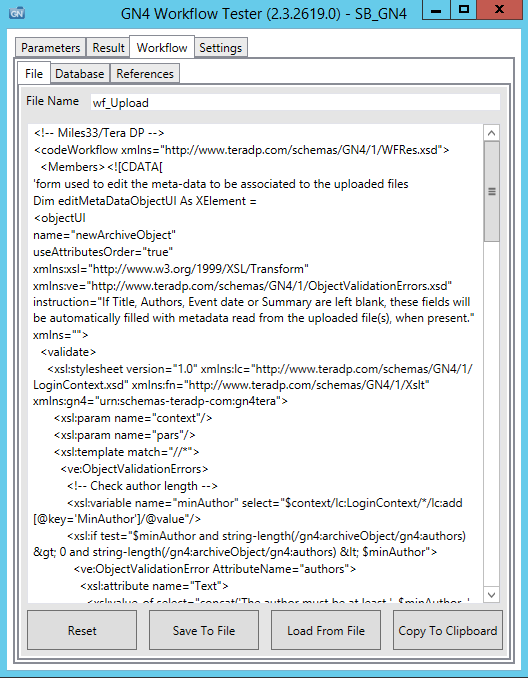
#### WorkflowTester – Workflow tab

Once the workflow is completed, if you want to run it into GN4, you have to create a ‘*wf\_[name].xml*’ file and load it into the GN4 database.

The ‘***Workflow-File***’ tab displays the final workflow generated by the VB.NET source code (*SequentialProgram.vb* file or *NavigatorProgram.vb*) that you are currently debugging.

*WorkflowTester* can generate the ‘*wf\_[name].xml*’ workflow file for you: just click on the ‘*Save To File*’ button.

The ‘*File Name*’ field contains the name of the workflow (with or without the ‘*wf\_*’ prefix and the ‘*.xml*’ extension).



Note that, in theory, you could modify the workflow content directly in dialog before saving (because the textbox which shows the final workflow is not read-only), but it is not advised.

In fact, because the final workflow is automatically generated from the VB.NET source file, your changes could be lost.

The advised way to change the workflow is:

1. stop the execution of *WorkflowTester* program
2. change the source .vb file using *VisualStudio Express* (*SequentialProgram.vb* file or *NavigatorProgram.vb*)
3. restart *WorkflowTester* program in *VisualStudio Express* for debugging the workflow

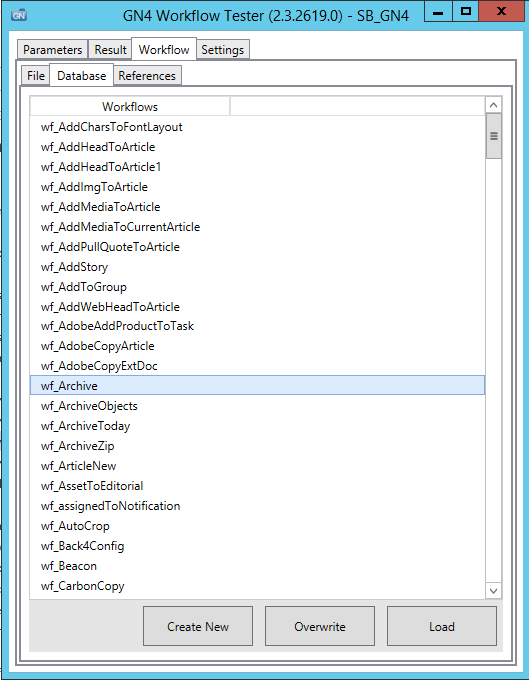
The ‘*Load From File*’ button tries to load the workflow from an existing ‘*wf\_[name].xml*’ file.

Note that, when loading the workflow file, *WorkflowTester.exe* program modifies the VB.NET source file (*SequentialProgram.vb* file or *NavigatorProgram.vb*) and closes itself. This way, restarting the *WorkflowTester.exe* program, you can debug the loaded workflow into *VisualStudio Express*.

The ‘*Reset*’ button clears the current source file (*SequentialProgram.vb* file or *NavigatorProgram.vb*) and closes the application, so that you can start to write a new workflow from scratch.

Note that, every time the source file is automatically changed, a backup copy is stored into the **workflowTesterBackup** folder: this way, you never lose your work.

Into the ‘***Workflow-Database***’ tab, you see the list of all the workflows stored into the database.



Selecting one workflow, you can load it into the application, clicking on the ‘*Load*’ button.

To overwrite an existing workflow into the database with your modified data, click the ‘*Overwrite*’ button.

Important: when overwriting an existing workflow into the database, a backup copy is automatically stored into the **workflowTesterBackup** folder.

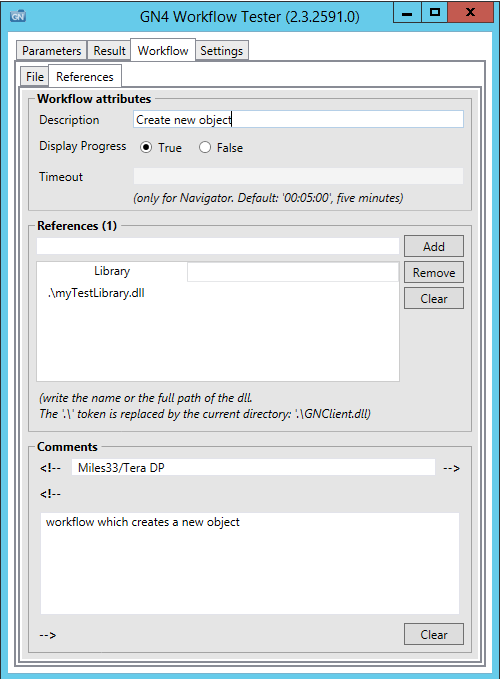
This way, if needed, you can easily restore the original workflow into the database.

The ‘*Create New’* button allows you to create a new workflow into the database. Note that you cannot specify the name of any already existing workflow.

Into the ‘***Workflow-References***’ tab you can set the *workflow* *attributes*, *references* and *comments*.

In *References* fields, to specify a local .dll which is stored into the same folder of the other standard GN4 libraries (like *Server.dll*, *Common.dll*, etc…) you have to write ***‘.\’*** as path.

For example:



GN4 automatically replaces *‘.\’* with the correct full path of the folder (that is, ‘**GN4\bin’** for *GNPortal*, ‘**I4\bin**’ for *I4*, ‘**Tera\bin’** for *Back4*, etc…). This trick is because, when *Back4* is started as windows service, it fails to look for and find the referenced assemblies into the *Back4.exe* folder, so we have to specify the full path of the library.

The *<References>* section of the workflow is automatically generated by the values inserted into the *References* list into the dialog.

Note that, on the other side, the *<Imports>* section of the workflow is automatically generated by the *Imports* statements written into the .vb source file (*SequentialProgram.vb* file or *NavigatorProgram.vb*).

To debug a workflow which uses a custom .dll:

1. put your custom .dll (called, for example, ‘*myTestLibrary.dll*’) into the ‘*WFTest\WorkflowTester\bin*’ folder
2. add to the project *WorkflowTesterExpress* the reference to the library *myTestLibrary.dll*:
   * right-click on the *WorkflowTesterExpress* project into the *SolutionExplorer* window of *VisualStudio Express*, select *Properties* and open the *References* tab. Click the ‘*Add’* button, then click on the ‘*Browse’* button and select the library file

The reference in *VisualStudio* is needed to debug the workflow.

1. run the WorkflowTester.exe program in *VisualStudio* (clicking *F5*).
2. write ‘.\*myTestLibrary.dll’* into the *References* field (see the above picture). This is needed if you want to generate the workflow file using the ‘*Save To File*’ button.
3. put a breakpoint at the first line of the workflow into the .vb source file (*SequentialProgram.vb* file or *NavigatorProgram.vb*).
4. click on the ‘Execute’ button and run the workflow

Into the ‘*Workflow-References*’ tab, we can set the attributes of the workflow tag ***(<Sequential>***or ***<Navigator>***). These attributes are:

* **Description**: it is a generic name of the workflow, used in log messages. For example, if the *Description* = “*Cropping*”, the final message will be “*Cropping completed successfully*. ”
* **DisplayProgress**: it is a flag which indicates if showing the progressing messages (or not) while the workflow is executing.

See the ‘[Progressing messages’](#_Progressing_messages) section into this document for more info about the workflow progress management**.**

* **Timeout**: it is the timeout of the workflow dialogs (the default is ’00:05:00’ = five minutes). It is valid only for the interactive (*Navigator*) workflows. It is needed to terminate the interactive workflow on server when the user leaves the workflow dialog open for many minutes on client, without performing any action.

Setting these attributes in the *WorkflowTeste*r application, and then saving the workflow file, the resulting workflow will be something like this:

<codeWorkflow

xmlns="http://www.teradp.com/schemas/GN4/1/WFRes.xsd">

<Navigator Description="Cropping" TimeOut="00:05:00" DisplayProgress="true">

<![CDATA[

. . .

]]>

</Navigator>

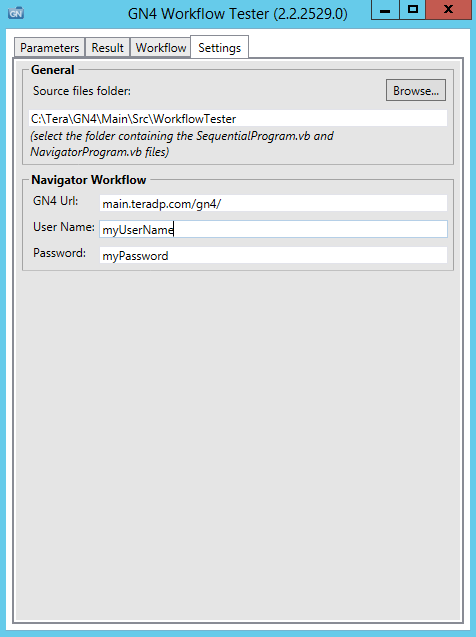
</codeWorkflow>

Into the ‘*Workflow-References*’ tab, there are two fields where to write textual comments that are copied at the beginning of the workflow file.

The comments are: a single general header (generally set to “*Miles33/Tera DP*”) and a multi-line text where you can describe what the workflow does and what parameters it expects.

#### WorkflowTester – Settings tab

The ‘***Settings***’ tab contains the general configuration of the *WorkflowTester* program.



The ‘*Source files folder*’ is the full path of the folder where are stored the .vb source files: *SequentialProgram.vb* file and *NavigatorProgram.vb*.

*WorkflowTester* tries to populate automatically this values; however, if it is wrong, you can select the correct path using the ‘*Browse*’ button.

The source full path is needed because *WorkflowTester* modifies the .vb source files when loads an existing workflow file (see the ‘*Load From File*’ button into the ‘*Workflow-File*’ tab).

The ‘*Navigator Workflow*’ fields are the client credentials needed to access to the GN4 server (server url, username and password). *WorkflowTester* uses these credentials to emulate a client application and show the visual controls in the workflow panel.

### Code workflow best practices:

#### Syntax statements

Here are some notes to keep in mind about the *CodeWorkflow* syntax (examples of these points are in other paragraphs of this document).

* *VB.NET* we use the *XElement* and *XDocument* classes to write/read the xml (here more info about their syntax: <http://msdn.microsoft.com/en-us/library/bb384460.aspx>). While in the old *WidowsWorkflows* we have to write the xml pieces out of the workflow at the bottom of the ‘*wf\_...*’ file, in *CodeWorkflows* the *XElement* class allow us to write xml tags directly into the code. Moreover, we can embed the value of variables into the xml using the “**<%= … %>**” syntax.
* *VB.NET* provides a way to manage the runtime exceptions: the **Try...Catch…** statement. This statement surrounds a piece of code: if an error occurs, the exception object is passed to the *Catch* code, where the exception can be managed as needed. Here more information about the *Try…Catch…* statement: <http://msdn.microsoft.com/en-us/library/fk6t46tz.aspx>
* In old *WindowsWorkflows*, the conversion from a number id (12345) to its string representation (‘obj12345’) and other similar conversions are applied into the XSLT stylesheet by the extension methods (namespace xmlns:fn="http://www.teradp.com/schemas/GN4/1/Xslt"). These utility methods are available also into the *CodeWorkflows*: they are in the **Context.XsltExtensions** property.
* To access the data of the current user you can use the **Context.LoginContext** property.

#### Progressing messages

While the workflow is running and the activities are executed one by one, the workflow panel can display *progressing messages*.

To enable the progressing messages, you have to set the **DisplayProgress** property of the workflow to ‘*true*’ (lowercase).

In sequential (batch) workflow:

<Sequential DisplayProgress="true">

In navigator (interactive) workflow:

<Navigator DisplayProgress="true">

Note that the default value is ‘*false*’; so, if the *DisplayProgress* attribute is missing, then the workflow panel does not display any progressing message.

Important: running Fiddler can prevent the workflow panel to display the progressing messages.

The progressing messages indicate what activity is actually running.

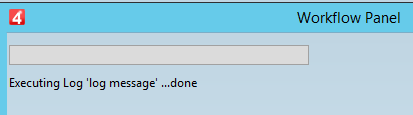
The *default* progressing message is: “ **Executing [*activity\_type*] ‘[*activity\_name*]’** ”

where [*activity\_type*] is the kind of the running activity (*Log*, *TransformXml*, *LoadObjects*, etc…) and [*activity\_name*] is the *Name* of the activity.

For example, this *Log* activity

Dim logAct As Log = New Log(Context) With {.Name = "log message"}

generates the following progressing message:

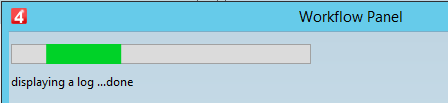


To customize the progressing message, you have to set the *Description* property of the activity.

For example, this *Log* activity

Dim logAct As Log = New Log(Context) With {.Name = "log message", .Description = "displaying a log"}

generates the following progressing message:



In *Navigator* workflows, also the interactive activities (like *Upload*, *Publish*, *Crop*, *Preview*, etc…) have the *DisplayProgress* property (this is a legacy from the old XAML *WindowWorkflows*).

Note that the *DisplayProgress* value of the activity can differ from the main workflow’s one. This allows user to enable/disable the progressing messages AFTER the execution of the interactive activity.

Into the following example, the progressing messages are displayed only AFTER the execution of the *Preview* activity, because the activity has DisplayProgress = True, while the main *Navigator* workflow has DisplayProgress="false":

<codeWorkflow xmlns="http://www.teradp.com/schemas/GN4/1/WFRes.xsd">

<Navigator DisplayProgress="false">

<![CDATA[

'first log

Dim beforeLogAct As Log = New Log(Context) With {.Name = "log message", .Description = "this is not displayed"}

beforeLogAct.Message = "before preview"

Dim beforeLogRes As LogResult = beforeLogAct.Do()

'preview

Dim previewAct As Preview = New Preview(Context) With {

.Name = "preview",

.DisplayProgress = True

}

previewAct.ObjectIds = Context.ObjectIds

Dim previewRes As PreviewResult = Await previewAct.Do()

'last log

Dim afterLogAct As Log = New Log(Context) With {.Name = "log message", .Description = "this is displayed"}

afterLogAct.Message = "after preview"

Dim afterLogRes As LogResult = afterLogAct.Do()

]]>

</Navigator>

</codeWorkflow>

#### Error management

When something’s wrong into a workflow at runtime, the .NET environment throws an exception: it stops the execution, creates a *System.Exception* object which contains all the information about the error (error message, memory stack, etc…), and makes this object available to the executing code.

It’s up to the code to choose if catching this exception and resuming the execution, or showing a message to the user, or let it go into a crash.

There are many standard types of exception objects (*System.Exception.ArgumentNullException*, *System.Exception.ArgumentOutOfRangeException*, etc…): every exception type indicates what error occurred.

GN4 has its own exceptions types. The exception type for the workflow is: **TeraDP.GN4.Common.WorkflowException**. So, if you want to notify GN4 that something specific of the workflow is wrong, you have to throw an exception of this type.

For example:

Dim destPath As String = Context.ParValue("destPath")

If String.IsNullOrEmpty(destPath) Then

Throw New TeraDP.GN4.Common.WorkflowException(TeraDP.GN4.Common.ErrorCode.WFInvalidParam, "destPath")

End If

In *CodeWorkflows* you can manage an error in different ways.

If you explicitly throw an exception, like in the example above, then the workflow stops its execution; but it is not the only choice you have.

You can logging the error as warning using a **Log** activity, and then continue the execution of the workflow:

If Context.Data.Count = 0 Then

Dim inputErrorAct As Log = New Log(Context) With {.Name = "input error"}

inputErrorAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Warning

inputErrorAct.Message = "Missing input picture file"

Dim inputErrorRes As LogResult = inputErrorAct.Do()

End If

The output is something like this: “*<WARNING> Missing input picture file*”

Or you can log it as blocking error and exit from the workflow:

If Context.Data.Count = 0 Then

Dim inputErrorAct As Log = New Log(Context) With {.Name = "input error"}

inputErrorAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Error

inputErrorAct.Message = "Missing input picture file"

Dim inputErrorRes As LogResult = inputErrorAct.Do()

Return 'exit from the workflow

End If

The output is something like this: “*<ERROR> Missing input picture file*”

Note that *Back4* considers a workflow failed (and eventually tries to re-execute it) when the output contains at least one log entry with ‘*Error’* code (LogCode.Error).

In the old *WindowsWorkflows* when a runtime exception occurs, the workflow stops. So, in *WindowsWorkflows* it is not possible to ignore an error and continue the execution.

On the other hand, because the *CodeWorkflows* are pieces of .NET code, they can catch the .NET exceptions. Infact, VB.NET provides a way to manage the runtime exceptions: the **Try...Catch…** statement.

This statement surrounds a piece of code: if an error occurs, the exception object is passed to the *Catch* code, where the exception can be managed as needed.

Here more information about the Try…Catch… statement: <http://msdn.microsoft.com/en-us/library/fk6t46tz.aspx>

Into the *Catch* code you can do anything you need: log the error message and continue

Try

Dim loadImgAct As LoadObjects = New LoadObjects(Context) With {.Name = "load picture"}

loadImgAct.ObjectIds = Context.ObjectIds

Dim loadImgRes As LoadObjectsResult = loadImgAct.Do()

Catch ex As Exception

'failed to load picture

Dim inputErrorAct As Log = New Log(Context) With {.Name = "load error"}

inputErrorAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Warning

inputErrorAct.Message = ex.Message

Dim inputErrorRes As LogResult = inputErrorAct.Do()

'continue the workflow execution

End Try

or return and exit from the workflow.

## Tutorials:

### Creating a new GN4 story into the database

We want to write a code workflow which receives a picture and creates a *gn4:story* object having the text “*Hello World!*” and containing the input picture.

First we have to open and setup the testing environment; then we try to compile the *CodeWorkflowTestExpress* project to check all the referenced dlls. When all is ready, we can start to write the code workflow.

The first thing to do is to pass the correct input data to the workflow. The workflow needs a picture file, so we can modify the **RunSequential** method of *Program.vb* module this way:

' Sequential workflow test call

hasLogError = WFTest.RunSequential(

GetType(SequentialWorkflow),

WFTest.Ids(),

WFTest.DataFiles("c:\temp\myPicture.jpg”),

WFTest.Pars(),

executionLog

)

*myPicture.jpg* is the image file that we want to insert into the new *gn4:story* object.

Let’s see all the pieces of the code workflow, one by one.

At the beginning of the workflow we want to test if all the needed input data are correctly set or not.

'check the input parameter

If Context.Data.Count = 0 Then

Throw New TeraDP.GN4.Common.WorkflowException("Missing input picture file")

End If

If no picture is passed to the workflow, then we throw an exception (see ‘*Error management*’ section).

When we are sure that the workflow has received picture files, then we can process them one-by-one.

In old *WindowsWorkflows* we need a *DataReplicator* activity to loop through a list of files. In CodeWorkflows we don’t need the *…Replicator* activities anymore because VB.NET has better statements, like **For Each**:

For Each data As TeraDP.GN4.Workflow.IActivityData In Context.Data

Next

First, into the loop, we want to save the input picture into the GN4 database. So, let’s write the *gn4:image* xml to pass to the **ImportXml** activity (like we do in the old *WindowsWorkflows*).

For Each data As TeraDP.GN4.Workflow.IActivityData In Context.Data

'create the picture xml

Dim imageXml As XElement =

<gn4:image name=<%= data.Info.SrcNameNoExtension & "\_" & Guid.NewGuid.ToString("N") %> xmlns:gn4="urn:schemas-teradp-com:gn4tera">

<gn4:folderRef><gn4:keyVal><%= Context.LoginContext.HomeFolderPath %></gn4:keyVal></gn4:folderRef>

<gn4:data mime="image/jpeg"><gn4:url><%= data.LocalPath %></gn4:url></gn4:data>

</gn4:image>

Next

Note that in VB.NET we use the *XElement* and *XDocument* classes to write/read the xml (here more info about their syntax: <http://msdn.microsoft.com/en-us/library/bb384460.aspx>).

While in the old *WidowsWorkflows* we have to write the xml pieces out of the workflow at the bottom of the ‘*wf\_...*’ file, in *CodeWorkflows* the *XElement* class allow us to write xml tags directly into the code.

Moreover, we can embed the value of variables into the xml using the “**<%= … %>**” syntax: very easy.

The name of the *gn4:image* is the name of the input file (without extension) and a unique suffix.

We can access the data of the current user via the *Context.LoginContext* property. In this example, we store the picture into the *home folder* of the current user (for the testing environment, it is “*system/users/Administrator*”).

Note the *mime-type* of the image data: for the moment, we assume that the input file is a **.jpeg** picture.

To create the image into the database, let’s pass the xml to the **ImportXml** activity.

Dim importImageAct As ImportXml = New ImportXml(Context) With {.Name = "import picture", .Description = "importing picture..."}

importImageAct.XmlIn = New XDocument(imageXml)

Dim importImageRes As ImportXmlResult = importImageAct.Do()

If importImageRes.CreatedIdsOut.Count = 0 Then

Throw New TeraDP.GN4.Common.WorkflowException(String.Format("Cannot create picture: '{0}'", data.LocalPath))

End If

Running this partial workflow, we should see the new image created into the GN4 database, but without ‘*thumbnail’* and ‘*preview’*, because no one has created them.

Note that the id of the newly created gn4:image is stored into the *importImageRes* object of type **ImportXmlResult**.

Once the picture is created, we have to create the story. Again, we write the *gn4:story* xml and pass it to the **ImportXml** activity.

Dim storyXml As XElement =

<gn4:story name=<%= Guid.NewGuid.ToString("N") %> xmlns:gn4="urn:schemas-teradp-com:gn4tera">

<gn4:folderRef><gn4:keyVal><%= Context.LoginContext.HomeFolderPath %></gn4:keyVal></gn4:folderRef>

<gn4:objs><gn4:ref idref=<%= Context.XsltExtensions.objectIdToString(importImageRes.CreatedIdsOut(0)) %>></gn4:ref></gn4:objs>

<gn4:xmlText><b>Hello World!</b></gn4:xmlText>

<gn4:xmlFormatRef><gn4:keyVal>XHTML</gn4:keyVal></gn4:xmlFormatRef>

</gn4:story>

importImageAct.XmlIn = New XDocument(storyXml)

importImageRes = importImageAct.Do()

If importImageRes.CreatedIdsOut.Count = 0 Then

Throw New TeraDP.GN4.Common.WorkflowException("Cannot create story")

End If

There is an important thing to note here: the id of the newly created *gn4:image* is stored into the *importImageRes.CreatedIdsOut* list as integer (for example, *12345*). To correctly write the *gn4:story* xml, we have to provide it into the string format, as “*obj12345*”.

In old *WindowsWorkflows*, this conversion (and other similar) is applied into the XSLT stylesheet by the extension methods (namespace xmlns:fn="http://www.teradp.com/schemas/GN4/1/Xslt").

These utility methods are available also into the *CodeWorkflows*: they are in the **Context.XsltExtensions** property. The call to *Context.XsltExtensions.objectIdToString(12345)* returns “*obj12345*”.

Running this partial workflow, we should see both the new image and the new story created into the GN4 database.

The ‘*thumbnail’* and ‘*preview*’ of a picture are created by the **Parse** activity (*ParseImg* plugin). In order to use the GN4 plugins, we need the parser dlls, so we have to copy the content of the **Parsers\_64.zip** daily archive into the *bin64* folder of the *CodeWorkflowTestExpress* installation.

Let’s process the input picture with the *Parse* activity:

Dim parseDataAct As Parse = New Parse(Context) With {.Name = "parse picture"}

parseDataAct.Data = data

parseDataAct.Options.PreviewSize = 200

parseDataAct.Options.ThumbnailSize = 80

Dim parseDataRes As ParseResult = parseDataAct.Do()

We set the *preview* and *thumbnail* size properties, so to have the binary of the generated images embedded into the output xmp (stored into the *parseDataRes.XmlOut property*) in *Base64* encoding.

So, in output xmp we have two *<xapGImg:image>* tags: the first contains the binary of the thumbnail, the second contains the binary of the preview.

We can read the images binary from the output xmp as strings, using the XElement.Descendants() method (which navigates the xml to find the tag name) and setting the correct *xapGImg* namespace.

Dim xapGImgNS As XNamespace = "http://ns.adobe.com/xap/1.0/g/img/"

Dim thumbnail As String = parseDataRes.XmlOut.Descendants(xapGImgNS + "image").FirstOrDefault.Value

Dim preview As String = parseDataRes.XmlOut.Descendants(xapGImgNS + "image")(1).Value

Moreover, we have the mime type of the input picture stored into the *parseDataRes.*MimeOut property.

Now we can modify the *gn4:image* xml to contain also the thumbnail and preview images.

Dim imageXml As XElement =

<gn4:image name=<%= data.Info.SrcNameNoExtension & "\_" & Guid.NewGuid.ToString("N") %> xmlns:gn4="urn:schemas-teradp-com:gn4tera">

<gn4:folderRef><gn4:keyVal><%= Context.LoginContext.HomeFolderPath %></gn4:keyVal></gn4:folderRef>

<gn4:data mime=<%= parseDataRes.MimeOut %>><gn4:url><%= data.LocalPath %></gn4:url></gn4:data>

<gn4:thumbnail mime="image/jpeg"><gn4:data><%= thumbnail %></gn4:data></gn4:thumbnail>

<gn4:preview mime="image/jpeg"><gn4:data><%= preview %></gn4:data></gn4:preview>

</gn4:image>

Now the workflow is ready. Running it into the testing environment, we should see both the new image and the new story created into the GN4 database: the story should contain the image and the image should have both the thumbnail and the preview correctly set.

Here is the final workflow:

'check the input parameter

If Context.Data.Count = 0 Then

Throw New TeraDP.GN4.Common.WorkflowException("Missing input picture file")

End If

For Each data As TeraDP.GN4.Workflow.IActivityData In Context.Data

Dim parseDataAct As Parse = New Parse(Context) With {.Name = "parse picture"}

parseDataAct.Data = data

parseDataAct.Options.PreviewSize = 200

parseDataAct.Options.ThumbnailSize = 80

Dim parseDataRes As ParseResult = parseDataAct.Do()

Dim xapGImgNS As XNamespace = "http://ns.adobe.com/xap/1.0/g/img/"

Dim thumbnail As String = parseDataRes.XmlOut.Descendants(xapGImgNS + "image").FirstOrDefault.Value

Dim preview As String = parseDataRes.XmlOut.Descendants(xapGImgNS + "image")(1).Value

'create the picture xml

Dim imageXml As XElement =

<gn4:image name=<%= data.Info.SrcNameNoExtension & "\_" & Guid.NewGuid.ToString("N") %> xmlns:gn4="urn:schemas-teradp-com:gn4tera">

<gn4:folderRef><gn4:keyVal><%= Context.LoginContext.HomeFolderPath %></gn4:keyVal></gn4:folderRef>

<gn4:data mime=<%= parseDataRes.MimeOut %>><gn4:url><%= data.LocalPath %></gn4:url></gn4:data>

<gn4:thumbnail mime="image/jpeg"><gn4:data><%= thumbnail %></gn4:data></gn4:thumbnail>

<gn4:preview mime="image/jpeg"><gn4:data><%= Preview %></gn4:data></gn4:preview>

</gn4:image>

Dim importImageAct As ImportXml = New ImportXml(Context) With {.Name = "import picture", .Description = "importing picture..."}

importImageAct.XmlIn = New XDocument(imageXml)

Dim importImageRes As ImportXmlResult = importImageAct.Do()

If importImageRes.CreatedIdsOut.Count = 0 Then

Throw New TeraDP.GN4.Common.WorkflowException(String.Format("Cannot create picture: '{0}'", data.LocalPath))

End If

Dim storyXml As XElement =

<gn4:story name=<%= Guid.NewGuid.ToString("N") %> xmlns:gn4="urn:schemas-teradp-com:gn4tera">

<gn4:folderRef><gn4:keyVal><%= Context.LoginContext.HomeFolderPath %></gn4:keyVal></gn4:folderRef>

<gn4:objs><gn4:ref idref=<%= Context.XsltExtensions.objectIdToString(importImageRes.CreatedIdsOut(0)) %>></gn4:ref></gn4:objs>

<gn4:xmlText><b>Hello World!</b></gn4:xmlText>

<gn4:xmlFormatRef><gn4:keyVal>XHTML</gn4:keyVal></gn4:xmlFormatRef>

</gn4:story>

importImageAct.XmlIn = New XDocument(storyXml)

importImageRes = importImageAct.Do()

If importImageRes.CreatedIdsOut.Count = 0 Then

Throw New TeraDP.GN4.Common.WorkflowException("Cannot create story")

End If

Next

The final step is to insert the workflow into a ‘*wf\_...*’ configuration file, load the file into the GN4 database and run the workflow in GN4.

1. create a xml file named ‘*wf\_CreateStoryAndImage.xml*’ and write the empty structure of the *CodeWorkflow*.

<codeWorkflow

xmlns="http://www.teradp.com/schemas/GN4/1/WFRes.xsd">

<References>

<!-- Add here references to additional assemblies (DLL) -->

<!-- <Reference>XXXX</Reference>-->

</References>

<Imports>

<!-- Add here additional namespaces to import -->

<!-- <Import>XXXX</Import> -->

</Imports>

<Members>

<![CDATA[

]]>

</Members>

<Sequential>

<![CDATA[

'empty workflow

]]>

</Sequential>

</codeWorkflow>

1. copy the workflow into the *CDATA* section inside the *<Sequential>* tag.
2. save the file
3. load the workflow into the database using *Cmd4.exe*:

**cmd4 config -in wf\_CreateStoryAndImage.xml**

1. run the workflow in GN4. For example using *Cmd4.exe*:

**cmd4 wf -name CreateStoryAndImage -in c:\temp\myPicture.jpg**

## Code workflow advanced training:

### Lesson One: load and save objects

The data of the objects stored into the GN4 database can be read exporting them as xml.

To see the xml of an object, we can call the ‘*objs*’ REST method in *GNPortal*, passing the id of the object as parameter. For example:

[http://localhost/GN4/do.ashx?cmd=**objs**&ids=1234](http://localhost/GN4/do.ashx?cmd=objs&ids=1234)

A GN4 object can contain references to other GN4 objects (for example, a *folderObject* contains the reference to its folder in the *folderRef* attribute).

To read the data of the referenced objects, we can *navigate* the xml: that is, we can expand the referenced objects into the xml, moving deeply into the xml structure, so to read more levels and to retrieve all the needed data.

To navigate the xml of an object, we can call the ‘*objsNav*’ REST method in *GNPortal*. For example:

[http://localhost/GN4/do.ashx?cmd=**objsNav**&ids=1234](http://localhost/GN4/do.ashx?cmd=objsNav&ids=1234)

How to do this in *CodeWorkflows*?

The activity which reads the xml of the GN4 objects if **LoadObjects**. This activity accepts a list of ids and returns the xml containing the object’s data.

For example, to read the xml of the object with id 1234, we can write:

Dim loadAct As LoadObjects = New LoadObjects(Context) With {.Name = "load object"}

loadAct.ObjectIds.Add(1234)

Dim loadRes As LoadObjectsResult = loadAct.Do()

If loadRes.XmlOut IsNot Nothing Then

loadRes.XmlOut.Save("c:\temp\myObject.xml")

End If

The pieces of workflow that run an activity follow always the same structure, that is:

* create the activity (New LoadObjects).
* set the parameters of the activity (in this case, the activity receives the object id *1234*).
* run the activity (loadAct.Do()): the result of the operation is returned into the ‘*…Result*’ object (LoadObjectsResult)
* read the result of the operation (in this case, the resulting xml is returned into the *XmlOut* property of *LoadObjectsResult*)

In the *CodeWorkflows*, the xml data are managed by the *XElement* and *XDocument* classes.

These are standard VB.NET classes. See <http://msdn.microsoft.com/library/bb387098.aspx> for more info about them.

Note that the *XDocument* class has the *Save* method, which allow us to easily store the xml into a local file.

Because *LoadObjectsResult.XmlOut* is a *XDocument*, we can write the instruction

loadRes.XmlOut.Save("c:\temp\myObject.xml")

and save the xml of the GN4 object into the file *myObject.xml*.

In general, saving the xml data to file is a good way to debug a *CodeWorkflow* in production environment.

Above we have seen how to read the data of GN4 objects retrieving their xml via the *LoadObjects* activity. But what if we want to create a new object into the GN4 database?

To create (or modify) a GN4 object, we can use the ‘import’ command of the *cmd4* tool, this way:

cmd4 import –in c:\temp\myObject.xml

This command parses the input xml and stores the read values into the database, creating or modifying GN4 objects.

Note that, if the xml contains the attributes needed to identify an object which is already existing into the database, then *cmd4* modifies that object; otherwise, a new object is created.

In *CodeWorkflows*, the activity which creates or modifies GN4 objects into the database is **ImportXml**.

Dim xmlData As XDocument = XDocument.Load("c:\temp\myObject.xml")

Dim importAct As ImportXml = New ImportXml(Context) With {.Name = "save object"}

importAct.XmlIn = xmlData

Dim importRes As ImportXmlResult = importAct.Do()

If importRes.IdsOut.Count > 0 Then

Dim logAct As Log = New Log(Context) With {.Name = "log message"}

logAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Info

logAct.Message = String.Format("Imported object with id: {0}", importRes.IdsOut(0))

Dim logRes As LogResult = logAct.Do()

End If

The sample above reads the file *myObject.xml* and imports it into the database. Finally, it logs the message with the id of the created or modified object.

### Lesson Two: xml data

Because GN4 returns and accepts object’s data as xml, it is important to know the VB.NET classes that manage the xml documents.

Two classes are extensively used in *CodeWorkflows*: ***XDocument*** and ***XElement***.

They are in the *System.Xml.Linq* .NET namespace (see <http://msdn.microsoft.com/en-us/library/system.xml.linq(v=vs.110).aspx> for more info about “*Linq To Xml*”).

The GN4 standard workflows in **GN4Config.zip** are a good place where to find many syntax examples.

The *XDocument* class represents an entire xml document, with xml declaration, etc…

It has the ***Load()*** and ***Save()*** methods that allow to easily read/write xml to file system.

For example:

Dim xmlData As XDocument = XDocument.Load("c:\temp\myObject.xml")

or

xmlData.Save("c:\temp\myObject.xml")

The *XElement* class is a generic node of an xml document. The *XDocument* class inherits from the *XElement* class.

In *CodeWorkflows* a *XElement* can be created simply writing the xml into the code, this way:

Dim myXElement As XElement =

<root>

<tag1>sample value</tag1>

</root>

Some properties of the *CodeWorkflows* activities (like the *TransformXml.XmlIn* property) require an *XDocument* object; other properties require a XElement object.

To create a XDocument from a XElement we can simply write:

Dim myXDocument As XDocument = New XDocument(myXElement)

Moreover, the **<%= … %>** syntax allow us to put the value of a VB.NET variable directly inside the xml. For example:

Dim objectId As Integer = 12345

Dim objectIdStr As String = String.Format("obj{0}", objectId)

Dim folderPath As String = "/system/users/Administrator"

Dim xImage As XElement =

<gn4:image id=<%= objectIdStr %> xmlns:gn4="urn:schemas-teradp-com:gn4tera">

<gn4:folderRef>

<gn4:keyVal><%= folderPath %></gn4:keyVal>

</gn4:folderRef>

</gn4:image>

xImage.Save("c:\temp\myImage.xml")

The “*Linq To Xml*”.NET classes provide many methods to look for specific data into an xml document.

For example, let’s assume that we have the following xml:

Dim xMyElement As XElement =

<gn4:image

id="obj12345" name="myImage" creationDate="2014-07-22T09:09:52.217Z" pixWidth="500" pixHeight="330"

xmlns:gn4="urn:schemas-teradp-com:gn4tera" xmlns:nav="http://www.teradp.com/schemas/GN4/1/ObjNav.xsd">

<gn4:folderRef idref="obj67890">

<nav:refObject idref="obj67890">

<gn4:folder nav:objectType="folder" id="obj67890" name="myFolder" path="/system/users/myFolder"/>

</nav:refObject>

</gn4:folderRef>

</gn4:image>

To find the name of the folder, we can use the **XPathSelectElement()** method:

Dim folderName As String = Nothing

Dim nsManager As XmlNamespaceManager = New XmlNamespaceManager(New NameTable())

nsManager.AddNamespace("gn4", "urn:schemas-teradp-com:gn4tera")

nsManager.AddNamespace("nav", "http://www.teradp.com/schemas/GN4/1/ObjNav.xsd")

Dim xFolder As XElement = xMyElement.XPathSelectElement("gn4:folderRef/nav:refObject/gn4:folder", nsManager)

If xFolder IsNot Nothing Then

folderName = xFolder.@name

End If

Once we have found the wanted *gn4:folder* tag, we can read its attribute using the ‘*.@*’ shorcut.

Note that, working with xml, it is always important to pay attention to the xml namespaces. In the above example we have to define both the *gn4* and *nav* prefixes to successfully read the folder tag.

Another method we can use is **Descendants**():

Dim gn4NS As XNamespace = "urn:schemas-teradp-com:gn4tera"

Dim xFolders As IEnumerable(Of XElement) = xMyElement.Descendants(gn4NS + "folder")

If xFolders IsNot Nothing AndAlso xFolders.Count > 0 Then

folderName = xFolders(0).@name

End If

There are also some shortcuts we can use:

...<node> = .Descendants("node")

.<node> = .Elements("node")

.@attr = .Attribute("attr").Value

Note that it is also possible to specify the xml namespace directly into the string parameter, using the curly brackets. For example:

xMyElement.Elements("{urn:schemas-teradp-com:gn4tera}folder")

Check the MSDN to find other ways to read a *XElement* in VB.NET.

Very often, in *CodeWorkflows*, we need to transform an xml using a stylesheet XSLT.

The activity which applies a XSLT to an xml is the **TransformXml**.

It receives the input xml and returns the transformed xml (note that, if the input xml is nothing, then the *TransformXml* activity returns the xml created from scratch by the stylesheet).

The XSLT to apply can be passed to the activity using a *XElement* embedded into the workflow, this way:

Dim transformAct As TransformXml = New TransformXml(Context) With {.Name = "apply xsl"}

transformAct.XmlIn = New XDocument(xMyElement)

transformAct.Xslt =

<xsl:stylesheet

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:msxsl="urn:schemas-microsoft-com:xslt"

xmlns:nav="http://www.teradp.com/schemas/GN4/1/ObjNav.xsd"

xmlns:gn4="urn:schemas-teradp-com:gn4tera"

version="1.0">

<xsl:template match="/\*">

<folder>

<xsl:value-of select="/gn4:image/gn4:folderRef/nav:refObject/gn4:folder/@name"/>

</folder>

</xsl:template>

</xsl:stylesheet>

Dim transformRes As TransformXmlResult = transformAct.Do()

Dim folderName As String = transformRes.XmlOut.Root.Value

The *TransformXml* activity can also accept the name of a stylesheet already loaded into the GN4 database.

Dim transformAct As TransformXml = New TransformXml(Context) With {.Name = "apply xsl"}

transformAct.XmlIn = New XDocument(xMyElement)

'load the xsl\_myStylesheet.xml config file stored into the database

transformAct.XsltConfigName = "myStylesheet"

Dim transformRes As TransformXmlResult = transformAct.Do()

### Lesson Three: files

Like in the old *WindowsWorkflows*, also in *CodeWorkflows* the files are wrapped by the **ActivityData** objects (or by its read-only interface **IActivityData**).

The *ActivityData* can contain the binary data as an array of bytes (*System.IO.Stream*) or as the file url (*ActivityData.Url* property).

Moreover, the *ActivityData* object has the **Info** property which exposes some file metadata (size, mime type, etc…)

In *CodeWorkflows,* to get the binary *Stream* from an *ActivityData* we can use the **Context.GetActivityDataStream**(*ActivityData*) method; vice versa, to create an *ActivityData* object from a binary stream we can use the static SequentialExecutionContext.**CreateActivityData**(*Stream*, …) method (see below for a syntax sample).

The activity which loads a binary file is the **Fetch** activity.

The *Fetch* activity accepts the url to fetch (supporting the ‘\*’ wildcard) and returns the list of found files (that is, the list of *ActivityData*).

It can read files also via FTP or HTTP. The *Fetch* activity exposes the properties where setting the web credentials (*UserName*, *Password* and *Domain*). However, if we don’t want to write the web credentials into the workflow, we can set them as login credentials (see the editing dialog of a gn4 user) or pass them to the workflow as parameters.

Let’s see, for example, how to read PDFs from a FTP folder:

Dim fetchAct As Fetch = New Fetch(Context) With {.Name = "read files"}

fetchAct.Url = "ftp://myFtpServer/myFolder/\*.pdf"

fetchAct.UserName = "username"

fetchAct.Password = "password"

Dim fetchRes As FetchResult = fetchAct.Do()

For Each data As TeraDP.GN4.Workflow.IActivityData In fetchRes.DataOut

Dim readPDFName As String = data.Info.SrcName

Next

In *CodeWorkflows* to save a file into a folder we have to use the **Save** activity.

The *Save* activity receives the file as *ActivityData* and the url where to save it. It can save files on FTP site, using the same web credentials management as the *Fetch* activity.

The following example loads a file from the file system (using .NET classes, but we could use the *Fetch* activity as well) and saves it into a FTP folder:

Dim destinationFullPath As String = Nothing

Dim data As TeraDP.GN4.Workflow.IActivityData = Nothing

Using fs As System.IO.FileStream = New System.IO.FileStream("c:\temp\test.pdf",

IO.FileMode.Open, IO.FileAccess.Read)

data = TeraDP.GN4.Server.SequentialExecutionContext.CreateActivityData(fs, "application/pdf", Nothing)

If data IsNot Nothing Then

Dim saveAct As Save = New Save(Context) With {.Name = "save file"}

saveAct.Data = data

saveAct.To = "ftp://myFtpServer/myFolder/test.pdf"

saveAct.UserName = "username"

saveAct.Password = "password"

Dim saveRes As SaveResult = saveAct.Do()

destinationFullPath = saveRes.ToOut

End If

End Using

If destinationFullPath IsNot Nothing Then

Dim logAct As Log = New Log(Context) With {.Name = "log message"}

logAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Info

logAct.Message = String.Format("Saved file: '{0}'", destinationFullPath)

Dim logRes As LogResult = logAct.Do()

End If

### Lesson Four: files and xml

Because the data of GN4 objects are read and saved as xml, in the *CodeWorkflows* we could need to read an xml physical file (*ActivityData* object) and transform it to a *XDocument* object, so to easily manage the data in VB.NET.

The activity which receives an xml file and returns an xml document is the **LoadXml** activity.

The *LoadXml* activity accepts an *ActivityData* (containing the binary or the url of an xml file) and returns an *XDocument*, which than we can pass to any activity accepting xml as input parameters (like the *TransformXml* or ImportXml activities).

In the following example, we load an xml physical file using the *Fetch* activity, and then transform it to an *XDocument*, so to read its xml data:

Dim fetchAct As Fetch = New Fetch(Context) With {.Name = "read file"}

fetchAct.Url = "ftp://myFtpServer/myFolder/test.xml"

fetchAct.UserName = "username"

fetchAct.Password = "password"

fetchAct.List = False 'download the binary from FTP

Dim fetchRes As FetchResult = fetchAct.Do()

For Each data As TeraDP.GN4.Workflow.IActivityData In fetchRes.DataOut

Dim fileToXmlAct As LoadXml = New LoadXml(Context) With {.Name = "file to xml"}

fileToXmlAct.Data = data

Dim fileToXmlRes As LoadXmlResult = fileToXmlAct.Do()

If fileToXmlRes.XmlOut IsNot Nothing Then

'read the name of the root tag

Dim rootTagName = fileToXmlRes.XmlOut.Root.Name.LocalName

End If

Next

If we want to do the opposite operation (that is, transforming a xml document to an xml file) in *CodeWorkflows*, we have to use the **SaveXml** activity.

The *SaveXml* activity accepts an *XDocument* and returns a file (*ActivityData* object), which than we can pass to any activity accepting files as input parameters (like the *Save* activity).

In the following example, we create an xml document from scratch and save it to a FTP folder using the *SaveXml* and *Save* activities:

'create an xml document

Dim xMyElement As XElement = <root><tag1>value1</tag1><tag2><tag3>value3</tag3>value2</tag2></root>

'convert the xml document to a file

Dim xmlToFileAct As SaveXml = New SaveXml(Context) With {.Name = "xml to file"}

xmlToFileAct.XmlIn = New XDocument(xMyElement)

xmlToFileAct.Encoding = "UTF-16"

xmlToFileAct.Indent = TeraDP.GN4.Common.XmlExportOptions.IndentValue.TwoSpaces

Dim xmlToFileRes As SaveXmlResult = xmlToFileAct.Do()

'save the file on FTP

Dim saveFileAct As Save = New Save(Context) With {.Name = "save file"}

saveFileAct.Data = xmlToFileRes.DataOut

saveFileAct.To = "ftp://myFtpServer/myFolder/test.xml"

saveFileAct.UserName = "username"

saveFileAct.Password = "password"

Dim saveRes As SaveResult = saveFileAct.Do()

'log the final path of file

Dim logAct As Log = New Log(Context) With {.Name = "log message"}

logAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Info

logAct.Message = String.Format("Saved file: '{0}'", saveRes.ToOut)

Dim logRes As LogResult = logAct.Do()

### Lesson Five: files in database

The GN4 database can contain binary data.

Actually, the binary data can be stored either into the database table or into a filesystem volume, depending how the ‘*dataContent*’ attribute is configured into the *.xsd* schema.

For example, the *gn4:image* object has its thumbnail stored directly into the database table, while the picture and the preview are stored into volumes.

To retrieve binary data from the GN4 database and make it available to other activities, we can use the **LoadData** activity.

The *LoadData* activity wants the id of the object to read and the name of its ‘*dataContent*’ attribute; it returns the found binary into an *ActivityData* object.

In the following example, we read the thumbnail of a *gn4:image* and save it into a folder:

Dim imageId As Integer = 12345

Dim loadThumbnailAct As LoadData = New LoadData(Context) With {.Name = "load thumbnail"}

loadThumbnailAct.ObjectIds.Add(imageId)

loadThumbnailAct.AttributeName = "thumbnail"

Dim loadThumbnailRes As LoadDataResult = loadThumbnailAct.Do()

'save the thumbnail on file

If loadThumbnailRes.DataOut.Count > 0 Then

Dim saveFileAct As Save = New Save(Context) With {.Name = "save file"}

saveFileAct.Data = loadThumbnailRes.DataOut(0)

saveFileAct.To = "c:\temp\thumbnail.jpg"

saveFileAct.Mode = Save.SaveMode.Overwrite

Dim saveRes As SaveResult = saveFileAct.Do()

'log the final path of file

Dim logAct As Log = New Log(Context) With {.Name = "log message"}

logAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Info

logAct.Message = String.Format("Saved file: '{0}'", saveRes.ToOut)

Dim logRes As LogResult = logAct.Do()

End If

If we want to do the opposite operation (that is, storing a binary file into a ‘*dataContent’* object’s attribute in the GN4 database), then we have to use the **StoreData** activity.

The *StoreData* activity accepts the id of the GN4 object, the name of its ‘*dataContent*’ attribute and the binary file to save (as *ActivityData* object).

In the following example, we load a picture from a folder and save it into the *thumbnail* attribute of a *gn4:image* object:

Dim imageId As Integer = 12345

Dim fetchAct As Fetch = New Fetch(Context) With {.Name = "read file"}

fetchAct.Url = "ftp://myFtpServer/myFolder/thumb.jpg"

fetchAct.UserName = "username"

fetchAct.Password = "password"

fetchAct.List = False 'download the binary from FTP

Dim fetchRes As FetchResult = fetchAct.Do()

'store the thumbnail into the database

For Each data As TeraDP.GN4.Workflow.IActivityData In fetchRes.DataOut

Dim saveThumbnailAct As StoreData = New StoreData(Context) With {.Name = "save thumbnail"}

saveThumbnailAct.ObjectIds.Add(imageId)

saveThumbnailAct.Data = data

saveThumbnailAct.MimeType = "image/jpg"

saveThumbnailAct.AttributeName = "thumbnail"

Dim saveThumbnailRes As StoreDataResult = saveThumbnailAct.Do()

Exit For

Next

### Lesson Six: file conversions

Usually, the users upload files of any kind in GN4 system (images, videos, audios, texts, file compressed, etc…). These files are imported into the database by a *CodeWorkflow* (*wf\_ImportActivityData.xml*) and the first thing the workflow must do is to recognize the type of each uploaded file.

For this purpose, the workflow uses the **Parse** activity.

The Parse activity is one of the three workflow activities that calls the GN4 plugins to perform their operations. The others are: *TransformData* activity and *Extract* activity (see below).

The order by which these three activities call the plugins is configured into the *web.config* (or *appSettings.xml*) file. For example:

<!-- Plugins configuration -->

<add key="Plugins.Parse" value="ParseArchive; ParseImg; ParseMultimedia; ParseAPDFL; ParseOffice; ParseCalendar; ParseText;" />

<add key="Plugins.Convert" value="ParseImg; ParseMultimedia; ParseOffice; ParseAPDFL; ParseCalendar; ParseText;" />

<add key="Plugins.Extract" value="ParseArchive; ParseAPDFL;" />

The *Parse* activity accepts the binary file to process, pass it to the GN4 plugins for recognizing it (*ParseImg*, *ParseMultimedia*, *ParseAPDFL*, etc…) and then returns an xml containing the file metadata (like the mime-type, the size, IPTC data, etc…).

Note that the metadata are written into the returned xml using the **ADOBE XMP** syntax (<http://www.adobe.com/devnet/xmp.html> ).

The uploading workflow reads these XMP metadata, creates the xml of the GN4 object and imports it into the database.

In the following example, we load all the files contained into a folder and find their mime-type.

'read the folder's files

Dim fetchAct As Fetch = New Fetch(Context) With {.Name = "read files"}

fetchAct.Url = "c:\myFolder\\*.\*"

Dim fetchRes As FetchResult = fetchAct.Do()

'process the files one by one

For Each data As TeraDP.GN4.Workflow.IActivityData In fetchRes.DataOut

Dim readFileName As String = data.Info.SrcName

'read the file's metadata

Dim parseAct As Parse = New Parse(Context) \_

With {.Name = "parse file", .Description = String.Format("parsing file '{0}'", readFileName)}

parseAct.Data = data

'parsing options: how to create thumbnails and xmp document

parseAct.Options.ThumbnailSize = 0

parseAct.Options.ThumbnailQuality = 0

parseAct.Options.ImageMinRes = 72

parseAct.Options.DefaultRes = 72

parseAct.Options.PreviewSize = 0

parseAct.Options.XmpRoot = False

parseAct.Options.UseAttributes = False

parseAct.Options.NoClipPreview = False

Dim parseRes As ParseResult = parseAct.Do()

Dim xmp As XDocument = parseRes.XmlOut

If parseRes.MimeOut IsNot Nothing Then

'read the mime type of the parsed file

Dim mime As String = parseRes.MimeOut

End If

Next

Note that the *Parse* activity accepts a list of parameters into the **ParseOptions** object. If the *ThumbnailSize* or *PreviewSize* parameters are greater than 0, then the binary of the generated thumbnails are returned embedded into the XMP metadata file (*Base-64* encoded). It is up to the importing XSLT stylesheet storing the binary of the thumbnails into the data content attributes of the GN4 object.

Many critical operations in GN4 need to transform the binary file, changing its format, size, etc…

For example, the printing process converts an xml file to pdf; the cropping process resizes a picture, and so on.

The activity which transforms a binary file is **TransformData**. It accepts the file to process and the list of conversion steps to perform on the file.

In the following example, we load a picture from the database, make it in grayscale, resize it, rotate it and finally save it into a file.

Dim imageId As Integer = 12345

'load the picture binary

Dim loadPictureAct As LoadData = New LoadData(Context) With {.Name = "load picture"}

loadPictureAct.ObjectIds.Add(imageId)

loadPictureAct.AttributeName = "data"

Dim loadPictureRes As LoadDataResult = loadPictureAct.Do()

If loadPictureRes.DataOut.Count > 0 Then

'transform the picture

Dim convertPictureAct As TransformData = New TransformData(Context) With {.Name = "convert picture"}

convertPictureAct.Data = loadPictureRes.DataOut.First

With convertPictureAct 'set the conversion steps

Dim stepRotate As TeraDP.GN4.Common.Step = New TeraDP.GN4.Common.Step()

stepRotate.Conversion = "Rotate"

stepRotate.Parameters.Add(-900) 'rotate 90° clockwise

Dim stepScale As TeraDP.GN4.Common.Step = New TeraDP.GN4.Common.Step()

stepScale.Conversion = "Scale"

stepScale.Parameters.Add(200) 'resize longer axis to 200 pixels

Dim stepColor As TeraDP.GN4.Common.Step = New TeraDP.GN4.Common.Step()

stepColor.Conversion = "ChangeColorSpace"

stepColor.Parameters.Add("GrayScale") 'move to grayscale

'the steps are execute in order

.Steps.Add(stepRotate)

.Steps.Add(stepScale)

.Steps.Add(stepColor)

End With

Dim convertPictureRes As TransformDataResult = convertPictureAct.Do()

If convertPictureRes.DataOut IsNot Nothing Then

'save the converted picture to file

Dim savePictureAct As Save = New Save(Context) With {.Name = "save picture"}

savePictureAct.Data = convertPictureRes.DataOut

savePictureAct.To = "c:\temp\converted.jpg"

savePictureAct.Mode = Save.SaveMode.Overwrite

Dim savePictureRes As SaveResult = savePictureAct.Do()

'log the final path of file

Dim logAct As Log = New Log(Context) With {.Name = "log message"}

logAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Info

logAct.Message = String.Format("Saved picture: '{0}'", savePictureRes.ToOut)

Dim logRes As LogResult = logAct.Do()

End If

End If

Beside parsing and converting the uploaded files, GN4 needs to extract files from the received compressed archives.

The activity which extracts files from a .zip archive is **Extract.**

To unzip a file the *Extract* activity calls the **ParseArchive** plugin.

The *Extract* activity is also used to generate the thumbnail pictures from a PDF file (**ParseAPDFL** plugin).

In the following example we decompress a .zip file into a folder:

Dim destinationFolder As String = "c:\temp"

'load the .zip file

Dim fetchAct As Fetch = New Fetch(Context) With {.Name = "load .zip file"}

fetchAct.Url = "c:\temp\myArchive.zip"

fetchAct.List = False

Dim fetchRes As FetchResult = fetchAct.Do()

For Each data As TeraDP.GN4.Workflow.IActivityData In fetchRes.DataOut

'decompress archive files (ZIP) extracting all the contained files

Dim extractAct As Extract = New Extract(Context) With {.Name = "extract"}

extractAct.Data.Add(data)

extractAct.Options.PreviewSize = 0

extractAct.Options.ThumbnailSize = 0

Dim extractRes As ExtractResult = extractAct.Do()

If extractRes.DataOut.Count > 0 Then

Dim logMessage As System.Text.StringBuilder = New System.Text.StringBuilder()

logMessage.AppendLine("Extracted files:")

For Each extractedData As TeraDP.GN4.Workflow.IActivityData In extractRes.DataOut

'save the decompressed file

Dim saveFileAct As Save = New Save(Context) With {.Name = "save file"}

saveFileAct.Data = extractedData

saveFileAct.To = String.Format("{0}\{1}", destinationFolder, extractedData.Info.SrcName)

saveFileAct.Mode = Save.SaveMode.Overwrite

Dim saveFileRes As SaveResult = saveFileAct.Do()

logMessage.AppendLine(saveFileRes.ToOut)

Next

'log the names of the unzipped files

Dim logAct As Log = New Log(Context) With {.Name = "log message"}

logAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Info

logAct.Message = LogMessage.ToString()

Dim logRes As LogResult = logAct.Do()

End If

Next

Another thing we can do playing with binary files is to create variations.

A variation can be any kind of conversion applied to the original file.

For example, the web publication generates different version of the image to publish, according to the sizes needed by the web site.

Almost all the objects in GN4 database can have variations (stored into the ‘*variants*’ attribute). For example, the archiving process creates PDF that are stored as variations of the *gn4:page* object.

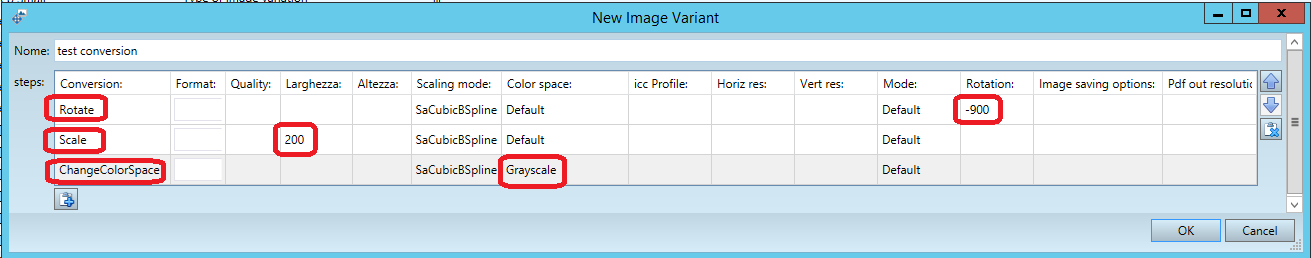
The activity which creates variations is **Variation**.

The *Variation* activity is mostly like the *TransformData* activity (both convert binary files) with the difference that the conversion steps are not explicitly written in the workflow, but they are read from the **variantType** object (the *variantTypes* are GN4 objects that contain a list of conversion steps).

Moreover, the *Variation* activity can store the generated binary files into the ‘*variants*’ attribute of the GN4 object, while the *TransformData* simply returns the files to the workflow.

In the following example, we create a binary variation using the *Variation* activity and applying the same conversion steps than the *TransformData* example above.

To run the example, we have to create in *EdAdmin4* (or in *GNPortal*) a new **variantTypeImage** object (called “*test conversion*”) specifying the conversion steps, like this:



Dim imageId As Integer = 12345

Dim variantTypeName As String = "test conversion"

'load the picture binary

Dim loadPictureAct As LoadData = New LoadData(Context) With {.Name = "load picture"}

loadPictureAct.ObjectIds.Add(imageId)

loadPictureAct.AttributeName = "low" 'it is an editorial image

Dim loadPictureRes As LoadDataResult = loadPictureAct.Do()

If loadPictureRes.DataOut.Count = 0 Then

'no picture found to convert

Dim logNoPictureAct As Log = New Log(Context) With {.Name = "log no picture"}

logNoPictureAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Warning

logNoPictureAct.Message = String.Format("Cannot find the binary of the picture (id: {0}).", imageId)

Dim logNoPictureRes As LogResult = logNoPictureAct.Do()

Return 'exit

End If

'apply the conversion

Dim createVariantAct As Variation = New Variation(Context) With {.Name = "create variant"}

createVariantAct.Data = loadPictureRes.DataOut

createVariantAct.ObjectId = imageId

createVariantAct.VariantTypeNames.Add(variantTypeName)

createVariantAct.Mode = Variation.CreationMode.CreateAlways

Dim createVariantRes As VariationResult = createVariantAct.Do()

'log the eventual errors in variant creation

Dim logVariantAct As Log = New Log(Context) With {.Name = "log variant result"}

logVariantAct.XmlMessages = createVariantRes.XmlOut

Dim logVariantRes As LogResult = logVariantAct.Do()

If createVariantRes.DataOut.Count > 0 Then

'save the converted picture to file

Dim savePictureAct As Save = New Save(Context) With {.Name = "save picture"}

savePictureAct.Data = createVariantRes.DataOut.FirstOrDefault

savePictureAct.To = "c:\temp\converted.jpg"

savePictureAct.Mode = Save.SaveMode.Overwrite

Dim savePictureRes As SaveResult = savePictureAct.Do()

'log the final path of file

Dim logAct As Log = New Log(Context) With {.Name = "log saved picture"}

logAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Info

logAct.Message = String.Format("Saved picture: '{0}'", savePictureRes.ToOut)

Dim logRes As LogResult = logAct.Do()

Else

'the variant has not been created

Dim logNoVariantAct As Log = New Log(Context) With {.Name = "log no variant"}

logNoVariantAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Warning

logNoVariantAct.Message = String.Format("Cannot create the variant of type '{0}' (id: {1}).", variantTypeName, imageId)

Dim logNoVariantRes As LogResult = logNoVariantAct.Do()

End If

Note that, unlike the *TransformData* example, here we have not to specify the conversion steps, because they are already configured inside the *variantTypeImage* “ test conversion”object

The resulting image is the same than the one created by the *TransformData*, but the *Variation* activity stores the binary into the ‘*variants*’ attribute of the GN4 object.

For example:

<img>

. . .

**<variants>**

<ref modifiedDate="2014-10-01T10:15:33.327Z" idref="obj3327">

<data id="data1428" mime="image/jpeg" size="8995">

<url>http://localhost/GN4/do.ashx?Cmd=Data&DataId=1428</url>

</data>

</ref>

**</variants>**

. . .

</img>

In case of errors, the *Variation* activity does not throw any exception, but it logs the error messages into an xml document (see the ‘*XmlOut*’ property). This is because, in publishing, we don’t want to stop the execution when the first error occurs. It’s up to the workflow to pass the logged xml to a *Log* activity (like in the example above), so to output the conversion results.

### Lesson Seven: call other workflows

Because a code workflow is a VB.NET script, it can enclose pieces of reusable code into *functions* and *subroutines*. Another way to avoid duplicating code is to create a sub-workflow which can be executed by other workflows.

A sub-workflow which can be called by other workflows must be *sequential* (that is, non-interactive) and must be stored into the database.

The activity which executes a sub-workflow is the **ExecuteSequentialWorkflow** activity.

This activity starts the sub-workflow (specified by the ‘*WorkflowName*’ property) passing the binary data (*Data* property), the ids of GN4 objects (*ObjectIds* property) and all the parameters needed by the sub-workflow to be executed.

Note that the parameters received by the main workflow are automatically available also in the sub-workflow.

In the following example, we call a sub-workflow which reads the text written into a *Word* document (*.docx* file). Note that, in this case, the read text is written by the called sub-workflow into its **ExecutionLog** xml, and it is available to the caller workflow in the ‘*ExecutionLog’* property of the activity.

'read the .docx file

Dim readWordFileAct As Fetch = New Fetch(Context) With {.Name = "read Word document"}

readWordFileAct.Url = "c:\temp\myFile.docx"

Dim readWordFileRes As FetchResult = readWordFileAct.Do()

If readWordFileRes.DataOut.Count = 0 Then

'no Word file found

Dim logNoFileAct As Log = New Log(Context) With {.Name = "log no file"}

logNoFileAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Warning

logNoFileAct.Message = String.Format("Cannot find the file '{0}'.", readWordFileAct.Url)

Dim logNoFileRes As LogResult = logNoFileAct.Do()

Return 'exit

End If

'convert .docx to html

Dim convertToHtmlAct As ExecuteSequentialWorkflow = New ExecuteSequentialWorkflow(Context) \_

With {.Name = "convert to html"}

convertToHtmlAct.WorkflowName = "DocxToHtml"

convertToHtmlAct.Data = New List(Of TeraDP.GN4.Workflow.IActivityData)({readWordFileRes.DataOut.First})

Dim convertToHtmlRes As ExecuteSequentialWorkflowResult = convertToHtmlAct.Do()

'the resulting html is stored into the ExecutionLog property

If convertToHtmlAct.ExecutionLog IsNot Nothing Then

'save the read text to file

Dim outputPath As String = "c:\temp\myFile.html"

convertToHtmlAct.ExecutionLog.Save(outputPath)

'success: log the final path of file

Dim logAct As Log = New Log(Context) With {.Name = "log saved file"}

logAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Info

logAct.Message = String.Format("Saved text: '{0}'", outputPath)

Dim logRes As LogResult = logAct.Do()

Else

'cannot read the file

Dim logFailedAct As Log = New Log(Context) With {.Name = "log failed"}

logFailedAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Error

logFailedAct.Message = String.Format("Cannot read the file content ('{0}').", readWordFileAct.Url)

Dim logFailedRes As LogResult = logFailedAct.Do()

End If

Note that the sub-workflow called by the *ExecuteSequentialWorkflow* activity can generate logs (*errors, warnings, information*). These logs are displayed at the end of the main workflow execution, just like the logs generated by the main workflow itself.

However, in some cases, it can be useful to verify directly inside the main workflow if the *ExecuteSequentialWorkflow* returned error logs or not.

In the following example, we use the **FirstError** property of the *ExecuteSequentialWorkflowResult* object to skip some action if the executed sub-workflow returned an error log:

Dim callSubWorkflowAct As ExecuteSequentialWorkflow = New ExecuteSequentialWorkflow(Context) \_

With {.Name = "call sub workflow"}

callSubWorkflowAct.WorkflowName = "mySubWorkflowName"

Dim callSubWorkflowRes As ExecuteSequentialWorkflowResult = callSubWorkflowAct.Do()

If callSubWorkflowRes.FirstError IsNot Nothing Then

'an error occurred into the called workflow

'...

Else

'no error occurred

'...

End If

### Lesson Eight: editing objects

When we want to show something to the user, we have to execute an interactive (*Navigator*) workflow.

Unlike the *Sequential* batch workflows, the Navigator workflows can execute visual activities (like *Upload*, *Crop*, *MultiCrop*, *Publish*, *SearchForm*, etc…)

The main difference between calling an interactive activity and a batch one is the **Await** word before the **.Do()**:

Dim editDialogAct As New EditObjects(Context) With {.Name = "edit dialog", .Title = "Modify objects"}

. . .

Dim editDialogRes As EditObjectsResult = Await editDialogAct.Do()

The *Await* word stops the workflow execution and waits for the user input. When the user clicks on the ‘*Next*’ button, the workflow execution is resumed.

The *Navigator* workflows are often used to edit the attributes of the GN4 objects, or to create a new GN4 object to import into the database.

The activity to create a new GN4 object is the **EditNewObject** activity.

The *EditNewObject* activity receives the xml of a single GN4 object and the xml of the **ObjectUI**.

The *ObjectUI* contains the list of the attributes to edit and the instructions about how to display the visual controls and the labels.

If no *ObjectUI* is set, then all the object’s attributes are prompted to the user for editing.

The *EditNewObject* activity does not store the edited object into the database. It simply returns the xml of the object, containing the values inserted by the user; then it’s up to the *ImportXml* activity, later in the workflow, to create the objects into the GN4 database.

In the following example, we show to the user a dialog where inserting the name and the folder of a new article object; then the new GN4 article is then created into the database.

'show the editing dialog

Dim createNewArticleAct As New EditNewObject(Context) \_

With {.Name = "create article", .Title = "Create article", .ShowFinishButton = True}

createNewArticleAct.ObjectsXml = <gn4:article xmlns:gn4="urn:schemas-teradp-com:gn4tera"/>

createNewArticleAct.ObjectUI =

<objectUI

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns="">

<default mode="Hidden"/>

<attribute name="name">

<ui mode="Normal" labelPosition="Above"/>

</attribute>

<attribute name="folderRef">

<ui mode="Normal" labelPosition="Above"/>

</attribute>

</objectUI>

Dim createNewArticleRes As EditNewObjectResult = Await createNewArticleAct.Do()

'read the inserted name

Dim xToImport As XDocument = New XDocument(createNewArticleRes.XmlOut)

Dim articleName As String = xToImport.Root.@name

Dim importArticleAct As ImportXml = New ImportXml(Context) With {.Name = "import article"}

importArticleAct.XmlIn = New XDocument(xToImport)

Dim importArticleRes As ImportXmlResult = importArticleAct.Do()

If importArticleRes.IdsOut.Count > 0 Then

'log the created id

Dim logAct As Log = New Log(Context) With {.Name = "log created"}

logAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Info

logAct.Message = String.Format("Created article '{0}' (id: {1})", articleName, importArticleRes.IdsOut.First.ToString())

Dim logRes As LogResult = logAct.Do()

End If

If we need to modify more objects at once, we have to use the **EditObjects** activity. The *EditObjects* activity accepts the ids of existing GN4 objects and the xml of the *ObjectUI*; it returns the full xml of the edited objects. Note that, unlike the *EditNewObject* activity, the *EditObjects* activity checks out the objects, verifying the access permissions of the user before showing the visual controls.

In the following example, we edit the description of two objects at the same time, saving the changes into the database:

Dim importXslt As XElement =

<xsl:stylesheet

version="1.0"

xmlns:gn4="urn:schemas-teradp-com:gn4tera"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:lc="http://www.teradp.com/schemas/GN4/1/LoginContext.xsd"

xmlns:oc="http://www.teradp.com/schemas/GN4/1/OperationContext.xsd"

xmlns:fn="http://www.teradp.com/schemas/GN4/1/Xslt">

<xsl:param name="pars"/>

<xsl:param name="context"/>

<xsl:template match="gn4:\*">

<xsl:element name="{local-name(.)}">

<xsl:attribute name="id">

<xsl:value-of select="@id"/>

</xsl:attribute>

<xsl:element name="description">

<xsl:value-of select="gn4:description"/>

</xsl:element>

</xsl:element>

</xsl:template>

<!-- Template matching a list of objects -->

<xsl:template match="gn4:objects">

<xsl:apply-templates/>

</xsl:template>

<!-- Root template -->

<xsl:template match="/">

<xsl:element name="objects">

<xsl:apply-templates/>

</xsl:element>

</xsl:template>

</xsl:stylesheet>

Dim objectIds As List(Of Integer) = New List(Of Integer)({3218, 3242})

'show the editing dialog

Dim editDialogAct As New EditObjects(Context) \_

With {.Name = "edit dialog", .Title = "Modify objects", .ShowFinishButton = True}

Try

editDialogAct.ObjectIds = objectIds

editDialogAct.ObjectUI =

<objectUI

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns="">

<default mode="Hidden"/>

<attribute name="description">

<ui mode="Normal" labelPosition="Above"/>

</attribute>

</objectUI>

Dim editDialogRes As EditObjectsResult = Await editDialogAct.Do()

'save changes into the database

Dim importAct As ImportXml = New ImportXml(Context) With {.Name = "import objects"}

importAct.XmlIn = New XDocument(editDialogRes.XmlOut)

importAct.Xslt = importXslt

Dim importRes As ImportXmlResult = importAct.Do()

If importRes.ModifiedIdsOut.Count > 0 Then

'log the modified ids

Dim logAct As Log = New Log(Context) With {.Name = "log modified"}

logAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Info

logAct.Message = String.Format("Modified {0} objects (ids: {1})", importRes.ModifiedIdsOut.Count,

TeraDP.GN4.GNClient.StringUtility.ListToString(importRes.ModifiedIdsOut))

Dim logRes As LogResult = logAct.Do()

End If

Finally

Dim uncheckOutAct As UndoCheckOut = New UndoCheckOut(Context) With {.Name = "uncheck out objects"}

uncheckOutAct.ActivityRef = editDialogAct.InternalName

Dim uncheckOutRes As UndoCheckOutResult = uncheckOutAct.Do()

End Try

There are two important things to note here.

First: because the *EditObjects* activity returns the full xml of the edited objects, we have to apply an *XSLT* stylesheet (called importXslt) to filter only the attributes that we want to change into the database. Otherwise, if we pass the full xml directly to the *ImportXml* activity, then all the attributes will be overwritten (also the binary ones).

Second: because the *EditObjects* activity checks out the GN4 objects, we need to uncheck out them before the workflow ends (otherwise, the objects will be automatically released when the user logs out).

The activity which unchecks out the objects locked by the workflow is the **UndoCheckOut** activity. The *UndoCheckOut* activity receives the unique internal name of the activity (*EditObjects*) which has checked out the objects.

The *UndoCheckOut* activity is placed into the **Finally** block, so to be sure to run it even if an exception occurs.

### Lesson Nine: search

To look for specific objects into the GN4 database, we have to perform a SQL query using **GNQuery**. *GNQuery* is the GN4 core module that converts a string (*XPath*) into the ‘SELECT…’ SQL statement.

See the **GNQuery.docx** document for many examples about how to find GN4 objects into the database using *GNQuery*.

Once we know what kind of object we need to find, to generate the correct *XPath* let’s image to navigate the xml returned by the **objsNav** REST command.

For example, let’s assume that we want to look for all the GN4 images that are into a folder with path ‘*/Wires/Images*’.

Navigating the xml of a *gn4:image* (4 levels deeply) <http://localhost/gn4/do.ashx?cmd=objsNav&ids=1234&max=4>

we have something like this:

<image id="obj1234" name="myImage">

. . .

<folderRef idref="obj3223">

<nav:refObject idref="obj3223">

<folder nav:objectType="folder" id="obj3223" name="Images" path="/Wires/Images" />

</nav:refObject>

</folderRef>

. . .

</image>

So, the correct *XPath* to check the folder’s path of the gn4:image is:

gn4:image[gn4:folderRef/nav:refObject/gn4:folder[@path='/Wires/Images']]

The activity which performs a SQL query into the GN4 database is the **Search** activity.

The *Search* activity accepts the *XPath* string and returns the ids of the found objects.

In the following example, we add a suffix to the name of all the images contained into the '*/Wires/Images*' folder:

Dim loadImagesXslt As XElement =

<xsl:stylesheet

version="1.0"

xmlns:gn4="urn:schemas-teradp-com:gn4tera"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:lc="http://www.teradp.com/schemas/GN4/1/LoginContext.xsd"

xmlns:oc="http://www.teradp.com/schemas/GN4/1/OperationContext.xsd"

xmlns:fn="http://www.teradp.com/schemas/GN4/1/Xslt">

<xsl:param name="pars"/>

<xsl:param name="context"/>

<xsl:template match="gn4:image">

<xsl:element name="{local-name(.)}">

<xsl:attribute name="id">

<xsl:value-of select="@id"/>

</xsl:attribute>

<xsl:attribute name="name">

<xsl:choose>

<xsl:when test="$pars and $pars/\*/add[@key='nameSuffix']">

<xsl:value-of select="concat(@name, $pars/\*/add[@key='nameSuffix']/@value)"/>

</xsl:when>

<xsl:otherwise>

<xsl:value-of select="@name"/>

</xsl:otherwise>

</xsl:choose>

</xsl:attribute>

</xsl:element>

</xsl:template>

<xsl:template match="gn4:objects">

<xsl:apply-templates/>

</xsl:template>

<xsl:template match="/">

<xsl:element name="objects">

<xsl:apply-templates/>

</xsl:element>

</xsl:template>

</xsl:stylesheet>

Dim folderPath As String = "/Wires/Images"

Dim nameSuffix As String = "\_mod"

'find the images ids

Dim searchAct As Search = New Search(Context) With {.Name = "search images"}

searchAct.XQuery = String.Format("gn4:image[gn4:folderRef/nav:refObject/gn4:folder[@path='{0}']]", folderPath)

Dim searchRes As SearchResult = searchAct.Do()

If searchRes.IdsCount = 0 Then

'cannot find images

Dim logNoImageAct As Log = New Log(Context) With {.Name = "log no image"}

logNoImageAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Warning

logNoImageAct.Message = String.Format("Cannot find any image in '{0}'.", folderPath)

Dim logNoImageRes As LogResult = logNoImageAct.Do()

Return 'exit

End If

'load the xml of the found images and append the suffix to the name attributes

Dim loadImagesAct As LoadObjects = New LoadObjects(Context) With {.Name = "load images"}

loadImagesAct.ObjectIds = searchRes.IdsOut

loadImagesAct.Xslt = loadImagesXslt

loadImagesAct.Pars.Add("nameSuffix", nameSuffix)

Dim loadImagesRes As LoadObjectsResult = loadImagesAct.Do()

'modify the images into the database

Dim importImagesAct As ImportXml = New ImportXml(Context) With {.Name = "import images"}

importImagesAct.XmlIn = loadImagesRes.XmlOut

Dim importImagesRes As ImportXmlResult = importImagesAct.Do()

If importImagesRes.IdsOut.Count > 0 Then

'log the modified ids

Dim logAct As Log = New Log(Context) With {.Name = "log modified"}

logAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Info

logAct.Message = String.Format("Modified images (ids: {0})",

TeraDP.GN4.GNClient.StringUtility.ListToString(importImagesRes.ModifiedIdsOut))

Dim logRes As LogResult = logAct.Do()

End If

If we need to search by an attribute containing a ‘ (*single quote*) character, then we have to specify it as *XQuery* parameter. For example:

Dim searchVideoAct As Search = New Search(Context) With {.Name = "search video"}

Dim strName As String = "friend's house"

searchVideoAct.Pars.Add("videoName", strName)

searchVideoAct.XQuery = "gn4:video[gn4:title=$videoName]"

Dim searchVideoRes As SearchResult = searchVideoAct.Do()

If we need to prompt the user for the query parameters, then we have to use the **SearchForm** interactive activity.

The *SearchForm* activity performs the query, shows the found objects and allows the user to select one or more of them. Finally, the selected ids are passed to the other activities of the workflow.

The following example is similar to the one above; but, this time, it’s up to the user to choose the folder where looking for images. Furthermore, only the selected images are modified into the database.

Dim nameSuffix As String = "\_mod"

'find the images ids

Dim searchFormAct As SearchForm = New SearchForm(Context) With {.Name = "search images"}

searchFormAct.SelectionMinCount = 1

searchFormAct.SearchObjectUI =

<searchObjectUI>

<attribute path="ObjectType.Id" op="In" objectTypes="image" mode="Hidden"/>

<attribute path="[image.folderRef]"/>

</searchObjectUI>

searchFormAct.DirectoryStyle =

<DirectoryStyle

Name="List"

Description=""

Height="1"

Width="-1"

xmlns:l="http://www.teradp.com/schemas/GN4/1/WinUI"

xmlns:xp="http://schemas.microsoft.com/winfx/2006/xaml/presentation">

<SortList>

<Sort Name="Name">

<SortDescription PropertyName="[folderObject.name]" IsAscending="true"/>

</Sort>

</SortList>

<ColStyle Name="Type" Width="30" Height="0">

<CellTemplate>

<xp:StackPanel Orientation="Horizontal">

<l:ObjectTypeIcon ObjectTypeName="group"/>

</xp:StackPanel>

</CellTemplate>

</ColStyle>

<ColStyle Name="Name" Width="150" Height="0">

<CellTemplate>

<xp:TextBlock HorizontalAlignment="Left" Text="{Binding Path=[folderObject.name]}"/>

</CellTemplate>

</ColStyle>

</DirectoryStyle>

Dim searchFormRes As SearchFormResult = Await searchFormAct.Do()

If searchFormRes.IdsOut.Count = 0 Then

Return 'exit

End If

'load the xml of the found images and append the suffix to the name attributes

Dim loadImagesAct As LoadObjects = New LoadObjects(Context) With {.Name = "load images"}

loadImagesAct.ObjectIds = searchFormRes.IdsOut

loadImagesAct.Xslt =

<xsl:stylesheet

version="1.0"

xmlns:gn4="urn:schemas-teradp-com:gn4tera"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:lc="http://www.teradp.com/schemas/GN4/1/LoginContext.xsd"

xmlns:oc="http://www.teradp.com/schemas/GN4/1/OperationContext.xsd"

xmlns:fn="http://www.teradp.com/schemas/GN4/1/Xslt">

<xsl:param name="pars"/>

<xsl:param name="context"/>

<xsl:template match="gn4:image">

<xsl:element name="{local-name(.)}">

<xsl:attribute name="id">

<xsl:value-of select="@id"/>

</xsl:attribute>

<xsl:attribute name="name">

<xsl:choose>

<xsl:when test="$pars and $pars/\*/add[@key='nameSuffix']">

<xsl:value-of select="concat(@name, $pars/\*/add[@key='nameSuffix']/@value)"/>

</xsl:when>

<xsl:otherwise>

<xsl:value-of select="@name"/>

</xsl:otherwise>

</xsl:choose>

</xsl:attribute>

</xsl:element>

</xsl:template>

<xsl:template match="gn4:objects">

<xsl:apply-templates/>

</xsl:template>

<xsl:template match="/">

<xsl:element name="objects">

<xsl:apply-templates/>

</xsl:element>

</xsl:template>

</xsl:stylesheet>

loadImagesAct.Pars.Add("nameSuffix", nameSuffix)

Dim loadImagesRes As LoadObjectsResult = loadImagesAct.Do()

'modify the images into the database

Dim importImagesAct As ImportXml = New ImportXml(Context) With {.Name = "import images"}

importImagesAct.XmlIn = loadImagesRes.XmlOut

Dim importImagesRes As ImportXmlResult = importImagesAct.Do()

If importImagesRes.IdsOut.Count > 0 Then

'log the modified ids

Dim logAct As Log = New Log(Context) With {.Name = "log modified"}

logAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Info

logAct.Message = String.Format("Modified images (ids: {0})",

TeraDP.GN4.GNClient.StringUtility.ListToString(importImagesRes.ModifiedIdsOut))

Dim logRes As LogResult = logAct.Do()

End If

### Lesson Ten: deleting objects

In GN4 it is possible to logically delete the objects (*spiking*). Once being spiked, the GN4 objects can be restored (*unspiking*) or deleted permanently (*purging*).

The activities that performs these three kinds of operations are: **Spike**, **Unspike** and **Purge** activities.

In the following example, we spike, unspike, spike again and finally purge a GN4 object:

'create an empty dummy object

Dim objectXml As XElement =

<gn4:document name="test object" xmlns:gn4="urn:schemas-teradp-com:gn4tera">

<gn4:folderRef><gn4:keyVal><%= Context.LoginContext.HomeFolderPath %></gn4:keyVal></gn4:folderRef>

<gn4:docTypeRef><gn4:keyVal>Unknown</gn4:keyVal></gn4:docTypeRef>

</gn4:document>

Dim importXmlAct As ImportXml = New ImportXml(Context) With {.Name = "create object"}

importXmlAct.XmlIn = New XDocument(objectXml)

Dim importXmlRes As ImportXmlResult = importXmlAct.Do()

If importXmlRes.IdsOut.Count <> 1 Then

'cannot create object

Dim logCreateAct As Log = New Log(Context) With {.Name = "log create"}

logCreateAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Error

logCreateAct.Message = "Cannot create empty object"

Dim logCreateRes As LogResult = logCreateAct.Do()

Return 'exit

End If

'spike the created object

Dim spikeAct As Spike = New Spike(Context) With {.Name = "spike object"}

spikeAct.ObjectIds = importXmlRes.IdsOut

Dim spikeRes As SpikeResult = spikeAct.Do()

If spikeRes.SpikedIds.Count <> 1 Then

'cannot spike object

Dim logSpikeAct As Log = New Log(Context) With {.Name = "log spike"}

logSpikeAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Error

logSpikeAct.Message = String.Format("Cannot spike object (id: {0})", importXmlRes.IdsOut.FirstOrDefault)

Dim logSpikeRes As LogResult = logSpikeAct.Do()

Return 'exit

End If

'un-spike the spiked object

Dim unspikeAct As Unspike = New Unspike(Context) With {.Name = "unspike object"}

unspikeAct.ObjectIds = spikeRes.SpikedIds

Dim unspikeRes As UnspikeResult = unspikeAct.Do()

If unspikeRes.UnspikedIds.Count <> 1 Then

'cannot un-spike object

Dim logUnspikeAct As Log = New Log(Context) With {.Name = "log un-spike"}

logUnspikeAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Error

logUnspikeAct.Message = String.Format("Cannot un-spike object (id: {0})", spikeRes.SpikedIds.FirstOrDefault)

Dim logUnspikeRes As LogResult = logUnspikeAct.Do()

Return 'exit

End If

'spike again the object

Dim spikeAgainAct As Spike = New Spike(Context) With {.Name = "spike again object"}

spikeAgainAct.ObjectIds = unspikeRes.UnspikedIds

Dim spikeAgainRes As SpikeResult = spikeAgainAct.Do()

If spikeAgainRes.SpikedIds.Count <> 1 Then

'cannot spike again the object

Dim logSpikeAgainAct As Log = New Log(Context) With {.Name = "log spike again"}

logSpikeAgainAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Error

logSpikeAgainAct.Message = String.Format("Cannot spike again object (id: {0})", unspikeRes.UnspikedIds.FirstOrDefault)

Dim logSpikeAgainRes As LogResult = logSpikeAgainAct.Do()

Return 'exit

End If

'purge the spiked object

Dim purgeAct As Purge = New Purge(Context) With {.Name = "purge object"}

purgeAct.ObjectIds = spikeAgainRes.SpikedIds

Dim purgeRes As PurgeResult = purgeAct.Do()

If purgeRes.PurgedIds.Count <> 1 Then

'cannot purge the object

Dim logPurgeAct As Log = New Log(Context) With {.Name = "log purge"}

logPurgeAct.Code = TeraDP.GN4.Workflow.LogEntry.LogCode.Error

logPurgeAct.Message = String.Format("Cannot purge object (id: {0})", spikeAgainRes.SpikedIds.FirstOrDefault)

Dim logPurgeRes As LogResult = logPurgeAct.Do()

Return 'exit

End If

## Activities:

Here is a very simple summary of the main activities input/output data:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Crop** | image | -> | crop | show the crop control (interactive) |
| **EditNewObject** | xml | -> | dialog | edit a GN4 object (interactive) |
| **EditObjects** | ids | -> | dialog | edit multiple GN4 objects (interactive) |
| **ExecuteSequentialWorkflow** |  | -> | workflow | call another workflow |
| ***Extract*** | zip | -> | files | decompress .zip file |
| ***Fetch*** | disk | -> | file | load a file |
| ***ImportXml*** | xml | -> | db | create/modify objects into the database |
| ***LoadData*** | db | -> | file | load binary data from the database |
| ***LoadObjects*** | db | -> | xml | load the objects xml from the database |
| ***LoadXml*** | file | -> | xml | convert a file into a xml document in memory |
| ***Log*** | string | -> | output | log a message |
| ***Parse*** | file | -> | metadata | extract metadata from a file |
| ***Purge*** | ids | -> | purge | purge the spiked objects |
| ***Save*** | file | -> | disk | save a file |
| ***SaveXml*** | xml | -> | file | convert an xml document into a file in memory |
| ***Search*** | xpath | -> | ids | perform a SQL query into the database |
| ***SearchForm*** | xpath | -> | ids | perform a SQL query into the database and show the found objects (interactive) |
| ***Spike*** | ids | -> | spike | spike objects |
| ***StoreData*** | file | -> | db | save binary data into the database |
| ***TransformData*** | file | -> | file | transform binary data |
| ***TransformXml*** | xml | -> | xml | apply an xslt stylesheet to an xml document |
| ***UndoCheckOut*** |  | -> | uncheck | release checked out objects |
| ***Unspike*** | ids | -> | unspike | unspike the spiked objects |
| ***Variation*** | file | -> | file | create a variant binary into the database |

Here is the list of the main activities and the description of their properties (the main input/output properties are highlighted):

### Crop (image -> crop)

Interactive activity. It shows the crop control to the user.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **ColumnWidth** | in | The page column width in millipoints | Integer |
| **DataIn** | in | The binary of the image to crop | Workflow.ActivityData |
| **DestinationIds** | in | The crop destinations ids. This list contains the ids of the ExtractDestinations to prompt to the user. Used only when the DestinationsXml property is empty. | List(Of Integer) |
| **DestinationNames** | in | The crop destinations names. This list contains the names of the ExtractDestinations to prompt to the user. Used only when the DestinationsXml property is empty. | List(Of String) |
| **DestinationsXml** | in | The ExtractDestinationList as XML node | XElement |
| **FixedDestinationName** | in | Fixed destination name | String |
| **ImgData** | in | The data (width, height) of the image | List(Of Integer) |
| **InterColumn** | in | The page column gutter in millipoints | Integer |
| **InterModule** | in | The page module gutter in millipoints | Integer |
| **ModuleHeight** | in | The page module height in millipoints | Integer |
| **OriginalName** | in | The name of the picture to crop | String |
| **ScaleCheck** | in | If true, the Scale Area checkbox is checked when the crop dialog starts; if false, the Scale Area checkbox is initially unchecked. | Boolean |
| **ShowFlip** | in | Show image flip commands | Boolean |
| **ShowPixel** | in | Show image size in pixel | Boolean |
| **ShowPrint** | in | Show image size in print unit | Boolean |
| **ShowRotate** | in | Show image rotation commands | Boolean |
| **TargetHeight** | in | The height of the target image to replace (dummy picture) in millipoints. Needed to display the wanted size of the crop frame. | Integer |
| **TargetWidth** | in | The width of the target image to replace (dummy picture) in millipoints. Needed to display the wanted size of the crop frame. | Integer |
| **Url** | in | The HTTP path of the image to crop | String |
| **CropDescription** | out | The crop description | String |
| **CropName** | out | The name of the cropped image (also in input). | String |
| **Flip** | out | If the image is flipped or not | Boolean |
| **FolderId** | out | Crop destination folder id | Integer |
| **HorizontalRes** | out | The wanted horizontal resolution of the cropped image in DPI | Integer |
| **MaxSize** | out | The maximum size of the image | Integer |
| **Overwrite** | out | If true, then overwrite existing object/file | Boolean |
| **RotateDegrees** | out | The image rotation in tenth of degree counterclockwise | Integer |
| **ScalePercentage** | out | The scaling percentage | Double |
| **SelectedDestDPI** | out | The resolution of the extractDestinations selected by the user (in DPI). | Integer |
| **SelectedDestId** | out | The id of the extractDestinations selected by the user returned as a single integer | Integer |
| **SelectedDestName** | out | The crop destination name selected by the user | String |
| **SelectedDestScale** | out | The scaling percentage to apply if we want to set the resolution of the extractDestinations selected by the user. | Double |
| **SelectedPreset** | out | The selected preset (if any) | Workfow.Preset |
| **UrlOut** | out | Composed file url | String |
| **UseDestResolution** | out | If true, then the workflow should scale the image and keep the resolution configured into the extractDestination. | Boolean |
| **Vertexes** | out | The cropping vertexes | List(Of Integer) |
| **VertexesString** | out | The cropping vertexes as string | String |
| **VerticalRes** | out | The wanted vertical resolution of the cropped image in DPI | Integer |

### EditNewObject (xml -> dialog)

Interactive activity. It edits a single new (i.e. not yet saved in the database) object. It displays the object fields into an editing form and returns the object xml with the values changed by the user.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **ObjectUI** | in | The serialized xml of the user interface options - i.e. how to edit the object | XElement |
| **ObjectsXml** | in | The serialized xml of the editing object. | XElement |
| **OverrideValues** | in | Indicates if the attributes that already have a value can be overridden (with the last used values) | Boolean |
| **ShowBackButton** | in | True if this activity must show the 'Back' button, in addition to the 'Next' one; False if only the 'Next' button is displayed. | Boolean |
| **ShowFinishButton** | in | True if this activity must show the 'Finish' button, instead of the 'Next' one; False if the 'Next' button is displayed. | Boolean |
| **Title** | in | The title of the dialog | String |
| **XmlOut** | out | The xml of the edited object | XElement |

### EditObjects (ids -> dialog)

Interactive activity. It edits one or more GN4 objects. It displays the objects fields into an editing form and returns the full xml of the edited object.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **ObjectUI** | in | The serialized xml of the user interface options - i.e. how to edit the object | XElement |
| **ObjectIds** | in | The ids of the GN4 objects to edit. | List(Of Integer) |
| **ShowBackButton** | in | True if this activity must show the 'Back' button, in addition to the 'Next' one; False if only the 'Next' button is displayed. | Boolean |
| **ShowFinishButton** | in | True if this activity must show the 'Finish' button, instead of the 'Next' one; False if the 'Next' button is displayed. | Boolean |
| **Title** | in | The title of the dialog | String |
| **XmlOut** | out | The full xml of the edited objects | XElement |

### ExecuteSequentialWorkflow ( -> workflow)

It executes a sub-workflow inside another workflow. It is useful - among other things - to share the code common to more than one workflow.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Data** | in | The binary data. | IEnumerable(Of Workflow.ActivityData) |
| **Emails** | in | Emails to process | IEnumerable(Of Common. Mail.EmailMessage) |
| **ExecutionLog** | in out | Log info concerning the workflow execution (typically list of files created) | XElement |
| **ObjectIds** | in | Ids of the objects to process. | List(Of Integer) |
| **ObjectIdVersions** | in | The object and object old version identifiers | List(Of GNClient. IdVersion) |
| **Pars** | in | List of generic parameters that are inserted into the xslt stylesheet as key/value pairs. The parameters value can be a string, a value convertible to string (e.g. integers), a list of integers, or XDocument/XElement.  To retrieve the parameters value, you can use the following XSL statement:  <xsl:value-of select=”$pars/\*/add[@key='mykey']/@value” /> | Dictionary(of String, Object) |
| **WorkflowName** | in | Name of the sequential workflow to be executed. Required. | String |
| **CreatedIdsOut** | out | Ids of all the objects created by the called workflow. | List(Of Integer) |
| **FirstError** | out | The first error log returned by the executed workflow. It is null if no error occurred. | Workflow.LogEntry |
| **IdsOut** | out | Ids of all the objects created or modified by the called workflow. | List(Of Integer) |
| **Logs** | out | The list of logs (errors, warnings, information) returned by the called workflow. It can be empty. | List(Of Workflow.LogEntry) |
| **ModifiedIdsOut** | out | Ids of all the objects modified by the called workflow. | List(Of Integer) |

### Extract (zip -> files)

It decompresses the zipped files. It is also used to generate the page thumbnails from a PDF file

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Data** | in | The list of files to process. | List(Of Workflow.ActivityData) |
| **Destination** | in | The temporary destination folder where to extract the compressed archives. | String |
| **MimeIn** | in | Supplementary MIME types to accept (as a comma-separated string). Used to decompress the .docx Office document. | String |
| **Options** | in | Parsing options.  See [here](http://tech.teradp.com/tech/html/gn4/docs/VSdoc/index.html#frlrfTeraDPXmpParseOptionsClassTopic_members--.html) for more information. | Common.ParseOptions |
| **DataOut** | out | The list of decompressed files. | List(Of Workflow.ActivityData) |
| **MimeOut** | out | MIME type resulted from the parsing | String |

### Fetch (disk -> file)

It retrieves files from the input url. The url to fetch can be a local path (for example *'c:\temp\\*.jpg*'), a HTTP or FTP url ('*http://mydomain/temp/\*.jpg'*) or a UNC path ('*\\myServer\temp\\*.jpg*')

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Back4Interface** | in | Set it to true if the passed url is a back4 http interface address. | Boolean |
| **Domain** | in | The domain to login to the web server. Optional. | String |
| **List** | in | If true (default), then simply lists the found files without actually download them; if false, then download the found files on server. Optional. | Boolean |
| **MaxBytes** | in | The maximum number of total file bytes (that is the sum of all the file sizes). Optional. Exceeding items are ignored. | Long |
| **MaxFiles** | in | The maximum number of found items. Optional. Exceeding items are ignored. | Integer |
| **MaxSize** | in | The maximum allowed size (in bytes). Optional. The bigger files are not considered. | Integer |
| **MinSize** | in | The minimum allowed size (in bytes). Optional. The smaller files are not considered. | Integer |
| **Password** | in | The password to login to the web server. Optional. | String |
| **Port** | in | The port to access to the server | Integer |
| **SingleWebFile** | in | If true, then the activity simply tries to download the single file specified by the input url without querying the folder for the list of files; if false (default), then the activity looks for the files into the url folder according to the file pattern. Optional. Set this flag to true to send a HTTP request to the input Url as is, without splitting it in folder-path and file-pattern parts. | Boolean |
| **ThrowExceptions** | in | If set to true the activity throws exceptions, else it fails silently. | Boolean |
| **Url** | in | The url of the files to read | String |
| **UserName** | in | The user name to login to the web server. Optional. | String |
| **DataOut** | out | The list of found files. | List(Of Workflow.ActivityData) |

### ImportXml (xml -> db)

It creates/modifies GN4 objects into database. It accepts the xml containing the attribute values of one or more GN4 objects.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **CheckedOutIds** | in | Database ids of the objects that have been checked-out persistently and that we are updating, and so should not be checked-out again. It can be empty if there are no such objects. | List(Of Integer) |
| **CopyAlways** | in | When importing data content from files always copy them into the destination volume, even if the file is in one of the defined volumes. | Boolean |
| **DefaultCategories** | in | Default categories for the resolution of category references. | Dictionary(Of String, String) |
| **DefaultObjects** | in | Default objects used when resolving references during import. | XElement |
| **IgnoreIds** | in | Ignore database ids in the XML. Useful when importing data coming from another GN4 system - and so with the same structure but with different database ids. | Boolean |
| **Overwrite** | in | Allow overwrite (modification) of existing objects. | Boolean |
| **SpikeChildren** | in | True to spike child objects that are no longer part of a parent object due to the modification caused by the import. | Boolean |
| **Trim** | in | A flag used for applying the trim operation if the examinated string is longer than the maxlength indicated on the schema. | Boolean |
| **TrimAppend** | in | Applying the trim operation, is the string to add at the end of the trimmed value to indicate that it has been cut. | String |
| **TrimChar** | in | Applying the trim operation, is the character to break the string. | String |
| **UseRefDefautObjs** | in | Use the DefaultObjects ref for every object reference (if false the DefaultObjects will be used only for resolving wrong references). | Boolean |
| **Validate** | in | Perform validation against the schema while importing. | Boolean |
| **XmlIn** | in | The xml to import. | XDocument |
| **Xslt** | in | Xsl transformation to apply to the source. If non-null it has precedence over XsltConfigName. | XElement |
| **XsltConfigName** | in | Name of the configuration object containing the XSLT. The specified name must NOT include the prefix 'xsl\_'. | String |
| **CreatedIdsOut** | out | Ids of all the objects created by the import operation. | List(Of Integer) |
| **IdsOut** | out | Ids of all the objects created or modified by the import operation. | List(Of Integer) |
| **ModifiedIdsOut** | out | Ids of all the objects modified by the import operation. | List(Of Integer) |

### LoadData (db -> file)

It reads the binary data (content attributes) of the GN4 objects.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **AttributeName** | in | The name of the attribute to load. Ignored if data ids are specified directly. Used only if DataIds is empty. | String |
| **DataIds** | in | The data ids to load. | List(Of Integer) |
| **ObjectIds** | in | Ids of the objects to read. | List(Of Integer) |
| **SrcPaths** | in | Optional source paths to assign to the loaded data (in the same order of the object ids or data ids) | List(Of String) |
| **DataOut** | out | The list of read files. | List(Of Workflow.ActivityData) |

### LoadObjects (db -> xml)

It reads the xml of GN4 objects. Used to retrieve the attribute values of one or more GN4 objects.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **AccessStrict** | in | True to make the export fail if any of the object attributes to be exported are not readable by the current user. If false (the default) such attributes would not be exported and the export would fail only if ALL the attributes are not readable by the current user. | Boolean |
| **CounterNames** | in | Names of the counters to increment for each exported object. | String() |
| **DataAsId** | in | True to export binary data only as id. | Boolean |
| **DataMaxSize** | in | Maximum size in bytes of the binary data to be inserted directly in the XML. Larger data is exported as URLs pointing back to the service.  If 0 all the data is exported as URL, if -1 all the data is inserted in the XML regardless of its size. Not used if DataAsId is true. | Integer |
| **Encoding** | in | Character encoding to use for the output. | String |
| **MaxDepth** | in | Maximum depth of the XML representation of the objects being exported. It is used when generating a feed, not when exporting stand-alone objects.  -1 = no limits (dangerous)  0 = generate stand-alone object XML, without navigation of the referenced objects. | Integer |
| **MimeType** | in | MIME type of the produced output. Useful when generating special XML formats (like RSS) that have their own MIME type. | String |
| **NavOptions** | in | Options controlling which extra information go in the XML. See [here](http://tech.teradp.com/tech/html/gn4/docs/VSdoc/index.html#frlrfTeraDPGN4GNClientDataNavigatorOptionClassTopic.html) for more information. | GNClient.DataNavigatorOption |
| **ObjectIdVersions** | in | The object and object old version identifiers | List(Of GNClient. IdVersion) |
| **ObjectIds** | in | Ids of the objects to read. | List(Of Integer) |
| **OmitXmlDeclaration** | in | True to omit the inital XML declaration in the output. | Boolean |
| **OutputNulls** | in | If True, attributes that have the special 'null' value - i.e. that are not present in the object - are still exported, either as xsi:nil elements or as empty attributes. | Boolean |
| **Pars** | in | List of generic parameters that are inserted into the xslt stylesheet as key/value pairs. The parameters value can be a string, a value convertible to string (e.g. integers), a list of integers, or XDocument/XElement.  To retrieve the parameters value, you can use the following XSL statement:  <xsl:value-of select=”$pars/\*/add[@key='mykey']/@value” /> | Dictionary(of String, Object) |
| **RefKeys** | in | Export key attribute(s) values of referenced objects. | Boolean |
| **Xslt** | in | Xsl transformation to apply to the source. If non-null it has precedence over XsltConfigName. | XElement |
| **XsltConfigName** | in | Name of the configuration object containing the XSLT. The specified name must NOT include the prefix 'xsl\_'. | String |
| **XmlOut** | out | The read xml. | XDocument |

### LoadXml (file -> xml)

It loads a xml or text file creating an in-memory XML - suitable to be used as input for XML-manipulation activities like *ImportXml*, *TransformXml* etc..

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Data** | in | The binary file to read | Workflow.ActivityData |
| **Encoding** | in | Encoding to use when reading from a plain text that does not have the standard initial bytes indicating its encoding. Default: “UTF-8”.  See [here](http://msdn.microsoft.com/en-us/library/system.text.encoding(v=vs.100).aspx) for more information. | String |
| **FromText** | in | true if the source is a plain text that should be converted to a simple XML with one element containing the whole text; false (default) if the source is an actual XML document. | Boolean |
| **PreserveWhitespace** | in | true if white space is preserved; false (default) white space is not preserved. | Boolean |
| **XmlOut** | out | The read xml. | XDocument |

### Log (string -> output)

It adds information to the log.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Code** | in | The logging code: info, warning, error...  See [here](http://tech.teradp.com/tech/html/gn4/docs/VSdoc/index.html#frlrfTeraDPGN4WorkflowLogEntryLogCodeClassTopic.html) for more information. | Workflow.LogEntry.LogCode |
| **Format** |  | The log entry format: Text, Html...  See [here](http://tech.teradp.com/tech/html/gn4/docs/VSdoc/index.html#frlrfTeraDPGN4WorkflowLogEntryLogFormatClassTopic.html) for more information. | Workflow.LogEntry.LogFormat |
| **Message** | in | The message to log | String |
| **XmlMessages** | in | The messages to log as xml serialized *LogEntry* classes. | XDocument |
| **MessageOut** | out | The composed final message. | String |

### Parse (file -> metadata)

It parses a file recognizing its format and extracting meta-data in XMP format (<http://www.adobe.com/products/xmp/>). XMP (*eXtensible Metadata Platform*) is the Adobe specification used to store many kinds of metadata as XML.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Data** | in | The binary file to parse | Workflow.ActivityData |
| **MimeOut** | in | MIME type resulted from the parsing. | String |
| **Options** | in | Parsing options for the XMP generation and XML representation.  See [here](http://tech.teradp.com/tech/html/gn4/docs/VSdoc/index.html#frlrfTeraDPXmpParseOptionsClassTopic_members--.html) for more information. | Common.ParseOptions |
| **XmlOut** | out | The xml containing the metadata in XMP format. | XDocument |

### Purge (ids -> purge)

It purges (that is, it removes them definitively) already spiked objects.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **By** | in | Number of objects to purge together in a single operation.  Greater the number quicker the operation is, and longer locks are put on the database. Optional, default value 50. | Integer |
| **Codes** | in | List of spike codes of the objects to purge.  Optional, if not specified objects spiked with any spike code will be purged. | Common.SpikeCode[] |
| **List** | in | If true then the activity will just return the ids of the objects that will be purged, instead than actually purging them. | Boolean |
| **ObjectIds** | in | Ids of the spiked objects to purge. | List(Of Integer) |
| **Since** | in | Time-span; e.g. 4:30 for 4 hours and 30 minutes, 2.06:00 for 2 days and 6 hours. Purge only the objects that have been spiked since the specified number of days / hours / minutes. Optional, if not specified only objects spiked more than 365 days ago will be purged. | TimeSpan |
| **Types** | in | List of object types to purge.  Optional, if not specified objects of any type will be purged. | String() |
| **PurgedIds** | out | The list of the ids of the purged objects | List(Of Integer) |
| **PurgedIdsString** | out | The ids of the purged objects as comma-separated string. | String |

### Save (file -> disk)

It creates a file into a folder. The file to save can be specified by a local path or by a web url.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Data** | in | The binary file to save | Workflow.ActivityData |
| **Domain** | in | The domain to login to the web server. Optional. | String |
| **HttpMethod** | in | The HTTP method to use in the saving request (POST or PUT). Default is PUT. | String |
| **Mode** | in | The saving mode: save only if missing, overwrite anyway or save with a unique name. See [here](http://tech.teradp.com/tech/html/gn4/docs/VSdoc/index.html#frlrfTeraDPGN4ServerCodeActivitySaveSaveModeClassTopic.html) for more information. | Server.CodeActivity. Save.SaveMode |
| **PassiveMode** | in | If true then uses the FTP passive-mode: otherwise it uses the FTP active-mode. | Boolean |
| **Password** | in | The password to login to the web server. Optional. | String |
| **Port** | in | The port to access to the server | Integer |
| **Timeout** | in | The number of seconds to wait for a FTP request.  To specify an infinite value, set the Timeout property to -1.  If it is set to 0, then the default value is used.  The default timeout appears to be 100 secs.  Used only for FTP requests. | Integer |
| **To** | In | The destination path. If set to a filename (and not a complete path), then the file is created with the specified name in the temp directory configured in ServerConfig.xml. If it is empty, then a file is created in the temp directory configured in ServerConfig.xml with a random name. | String |
| **UseLoginTemp** | in | If true and no explicit output name is specified save the data to a temporary file associated with the current login - i.e. that is removed only at logout (or when the login expires) | Boolean |
| **UserName** | in | The user name to login to the web server. Optional. | String |
| **ToOut** | out | Composed output file name. | String |

### SaveXml (xml -> file)

It receives a xml and simply converts it into a binary data (that is, create a temporary file)

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Encoding** | in | The file encoding. Ignored if the xsl stylesheet is specified. | String |
| **Indent** | in | Indentation. Ignored if the xsl stylesheet is specified. See [here](http://tech.teradp.com/tech/html/gn4/docs/VSdoc/index.html#frlrfTeraDPGN4CommonXmlExportOptionsIndentValueClassTopic.html) for more information. | Common.XmlExportOptions.  IndentValue |
| **NewLine** | in | New line characters to use.  See [here](http://tech.teradp.com/tech/html/gn4/docs/VSdoc/index.html#frlrfTeraDPGN4CommonXmlExportOptionsNewLineValueClassTopic.html) for more information. | Common.XmlExportOptions.  NewLineValue |
| **OmitXmlDeclaration** | in | Omit the inital XML declaration in the output. Ignored if the xsl stylesheet is specified. | Boolean |
| **Pars** | in | List of generic parameters that are inserted into the xslt stylesheet as key/value pairs. The parameters value can be a string, a value convertible to string (e.g. integers), a list of integers, or XDocument/XElement.  To retrieve the parameters value, you can use the following XSL statement:  <xsl:value-of select=”$pars/\*/add[@key='mykey']/@value” /> | Dictionary(of String, Object) |
| **SrcPath** | in | Optional source path to associate with the output data | String |
| **XmlIn** | in | The xml to read. | XDocument |
| **Xslt** | in | Xsl transformation to apply to the source. If non-null it has precedence over XsltConfigName. | XElement |
| **XsltConfigName** | in | Name of the configuration object containing the XSLT. The specified name must NOT include the prefix 'xsl\_'. | String |
| **DataOut** | out | The binary data. | Workflow.ActivityData |

### Search (xpath -> ids)

It performs a search into the database and returns the found object ids.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Pars** | in | List of generic parameters as key/value pairs used into the XQuery string. The parameters value can be a string, a value convertible to string (e.g. integers), a list of integers, etc…. | Dictionary(of String, Object) |
| **SpikedHandling** | in | A flag which indicates if returning the spiked GN4 objects or not.  The allowed values are:  - Exclude: find only normal (non-spiked) objects. This is the default.  - Include: find both spiked and non-spiked objects.  - Only: find only spiked objects. | Common.SpikedHandling |
| **XQuery** | in | The XPath string describing the search to execute | String |
| **IdsCount** | out | Number of found ids | Integer |
| **IdsOut** | out | Ids of all the objects found in the database | List(Of Integer) |
| **IdsOutString** | out | The found ids as comma-separated string. | String |

### SearchForm (xpath -> ids)

Interactive activity. It prompts user for the query parameters, performs the search into the database and shows the found objects.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **DirectoryStyle** | in | The serialized xml of the directory style - i.e. how to show the results | XElement |
| **MultipleSelection** | in | True if the multiple selection is allowed, false otherwise. Default is true. | Boolean |
| **SearchContext** | in | Search Context - i.e. how to search the object in i4 - used in the I4 interface only | String |
| **SearchExtra** | in | Search Extra - extra search which is anded with the search query. - used in the I4 interface | String |
| **SearchExtraPars** | in | The name-value collection containing Search Extra Pars - extra search pars for the search extra. - used in the I4 interface | System.Collections.Specialized.  NameValueCollection |
| **SearchObjectUI** | in | The serialized xml of the search interface options - i.e. how to search the object | XElement |
| **SelectionMinCount** | in | Minimum number of items that must be selected in the result list.  Default is 0, which means "no minimum". | Integer |
| **SelectionMaxCount** | in | Maximum number of items that must be selected in the result list.  Default is 0, which means "no maximum". | Integer |
| **IdsOut** | out | Ids of the selected GN4 objects returned by the search | List(Of Integer) |
| **IdsOutString** | out | The selected ids as comma-separated string | String |
| **ResultIdsOut** | out | All the ids of the GN4 objects returned by the search (not only the selected ones) | List(Of Integer) |
| **ResultIdsOutString** | out | All the found ids as comma-separated string (not only the selected ones) | String |
| **SearchConditionsXml** | out | The xml of the generated SearchConditions | XElement |

### Spike (ids -> spike)

It spikes GN4 objects (that is, it deletes them logically).

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Extend** | in | Comma-separated list of attribute types - specified as <object type name>.<attribute name>.  Optional. Spike also all the objects that reference the ones matching the condition via the attributes in this list. | String |
| **ExtendUnref** | in | Comma-separated list of attribute types - specified as <object type name>.<attribute name>.  Either spike or un-reference the objects that reference the specified ones via the attributes in this list - depending on the corresponding setting in the objects partition. | String |
| **ObjectIds** | in | Ids of the objects to spike. | List(Of Integer) |
| **Purge** | in | Specifies for how long the objects will be kept in the system before they are purged (permanently deleted).  If not specified the objects will be kept for the default time defined at system level.  Example: 4:30 for 4 hours and 30 minutes, 2.06:00 for 2 days and 6 hours. | TimeSpan |
| **Recurse** | in | If true, spike also the object that reference the ones matching the condition indirectly via the attributes specified by the Extend property (reference a referencing object etc.). Ignored if Extend is null. | Boolean |
| **SpikeCode** | in | Code specifying the type of spiking.  Possible values: Normal (standard spiking), Black (black copy) and Auto (automatic spiking of sub-objects, no longer used). | Common.SpikeCode |
| **StopOnFirstError** | in | If true, stop when the first error occurs  . | Boolean |
| **Unref** | in | Comma-separated list of attribute types - specified as <object type name>.<attribute name>.  Optional. Remove all the objects to be spiked from these reference (single or multiple) attributes. | String |
| **Errors** | out | The list of occurred exceptions. | List(Of Exception) |
| **SpikedIds** | out | The list of the ids of the spiked objects | List(Of Integer) |
| **SpikedIdsString** | out | Returns the ids of the spiked objects as comma-separated string. | String |

### StoreData (file -> db)

It stores the binary data (content attributes) of the GN4 objects into the database. It needs the name of the content attribute and the ids of the objects to update.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **AttributeName** | in | The name of the attribute where to store the file. | String |
| **CheckedOutIds** | in | Database ids of the objects that have been checked-out persistently and that we are updating, and so should not be checked-out again. It can be null or empty if there are no such objects. | List(Of Integer) |
| **Data** | in | The binary file to save | Workflow.ActivityData |
| **MimeType** | in | Optional mime-type string forced in the input file data. If null, the mime-type must be already set into the ActivityData.Info.SrcMime property. | String |
| **ObjectIds** | in | Ids of the objects to update. | List(Of Integer) |

### TransformData (file -> file)

It transforms the input file following the configured steps. It can, for example, convert a video to a different format, resize a picture, set the IPTC headers, etc...

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Data** | in | The binary file to convert | Workflow.ActivityData |
| **SrcPath** | in | Optional source path to assign to the output binary. | String |
| **Steps** | in | Converting options. They are a list of conversion Step.  See [here](http://tech.teradp.com/tech/html/gn4/docs/VSdoc/index.html#frlrfTeraDPXmpStepParametersClassTopic_members--.html) for more information. | List(Of Common.Step) |
| **DataOut** | out | The binary data. | Workflow.ActivityData |

### TransformXml (xml -> xml)

It applies an XSL transformation to the input XML and return the result.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Pars** | in | List of generic parameters that are inserted into the xslt stylesheet as key/value pairs. The parameters value can be a string, a value convertible to string (e.g. integers), a list of integers, or XDocument/XElement.  To retrieve the parameters value, you can use the following XSL statement:  <xsl:value-of select=”$pars/\*/add[@key='mykey']/@value” /> | Dictionary(of String, Object) |
| **XmlIn** | in | The xml to import. | XDocument |
| **Xslt** | in | Xsl transformation to apply to the source. If non-null it has precedence over XsltConfigName. | XElement |
| **XsltConfigName** | in | Name of the configuration object containing the XSLT. The specified name must NOT include the prefix 'xsl\_'. | String |
| **XmlOut** | out | Produced XML. | XDocument |

### UndoCheckOut (-> uncheck)

It undoes check-out objects checked-out by a specified activity.

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **ActivityRef** | in | The unique internal name of the activity which has checked out the objects. | String |

### Unspike (ids -> unspike)

It un-spikes objects (that is, it reverts them from logically deleted to normal).

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Extend** | in | Comma-separated list of attribute types - specified as <object type name>.<attribute name>.  Optional. Un-spike also all the objects that reference the ones matching the condition via the attributes in this list. | String |
| **ObjectIds** | in | Ids of the objects to un-spike. | List(Of Integer) |
| **Recurse** | in | If true, un-spike also the object that reference the ones matching the condition indirectly via the attributes specified by the Extend property (reference a referencing object etc.). Ignored if Extend is null. | Boolean |
| **Unref** | in | Comma-separated list of attribute types - specified as <object type name>.<attribute name>.  Optional. Remove all the objects to be un-spiked from these reference (single or multiple) attributes. | String |
| **UnspikedIds** | out | The list of the ids of the un-spiked objects | List(Of Integer) |
| **UnspikedIdsString** | out | The ids of the un-spiked objects as comma-separated string. | String |

### Variation (file -> file)

It creates a new binary variation of a GN4 object

|  |  |  |  |
| --- | --- | --- | --- |
| *Properties* |  | *Description* | *VB.NET type* |
| **Data** | in | The binary file to convert | IEnumerable(Of Workflow.ActivityData) |
| **AttributeName** | in | The name of the attribute to modify (default: 'data'). It can be empty if the input Data property is set and the Mode is 'CreateTemp' and the 'CreateAlways' property is false. | String |
| **CheckedOutIds** | in | Database ids of the objects that have been checked-out persistently and that we are updating, and so should not be checked-out again. It can be empty if there are no such objects. | List(Of Integer) |
| **CreateAlways** | in | It indicates if overwriting the old variations despite their last modified date. If true, it always creates and stores the variation into GN4 database, overwriting the old variations. If false, the variation is generated according to the 'Mode' property. (Left for retro compatibility) | Boolean |
| **Mode** | in | How to create and store the binary variation. It can be: "CreateAlways", "CreateNew" (default) or "CreateTemp".  See [here](http://tech.teradp.com/tech/html/gn4/docs/VSdoc/index.html#frlrfTeraDPGN4ServerCodeActivityVariationCreationModeClassTopic.html) for more information. | CodeActivity.Variation.  CreationMode |
| **ObjectId** | in | The id of the ‘folderObject’ for which variation files must be created. It can be empty if the input Data property is set and the Mode is 'CreateTemp' and the 'CreateAlways' property is false. | Integer |
| **SrcPaths** | in | Optional source paths to assign to the output variations (in order of creation) | List(Of String) |
| **VariantTypeIds** | in | The list of the ids of the variantType to look for. | List(Of Integer) |
| **VariantTypeNames** | in | The list of the names of the variantType to look for.  This property is ignored if VariantTypeIds is not empty. | List(Of String) |
| **DataOut** | out | Converted data. | List(Of Workflow.ActivityData) |
| **XmlOut** | out | Produced XML | XDocument |