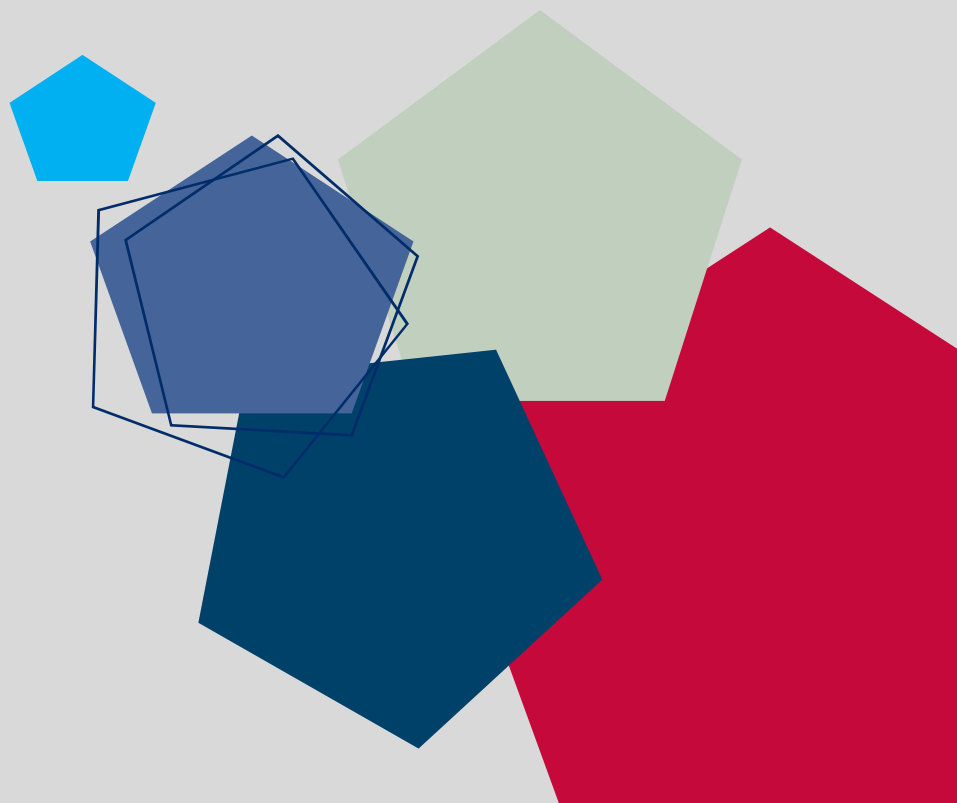


Automating Custom SSIS Tasks in Biml

© 2019 Ben Weissman / Tillmann Eitelberg



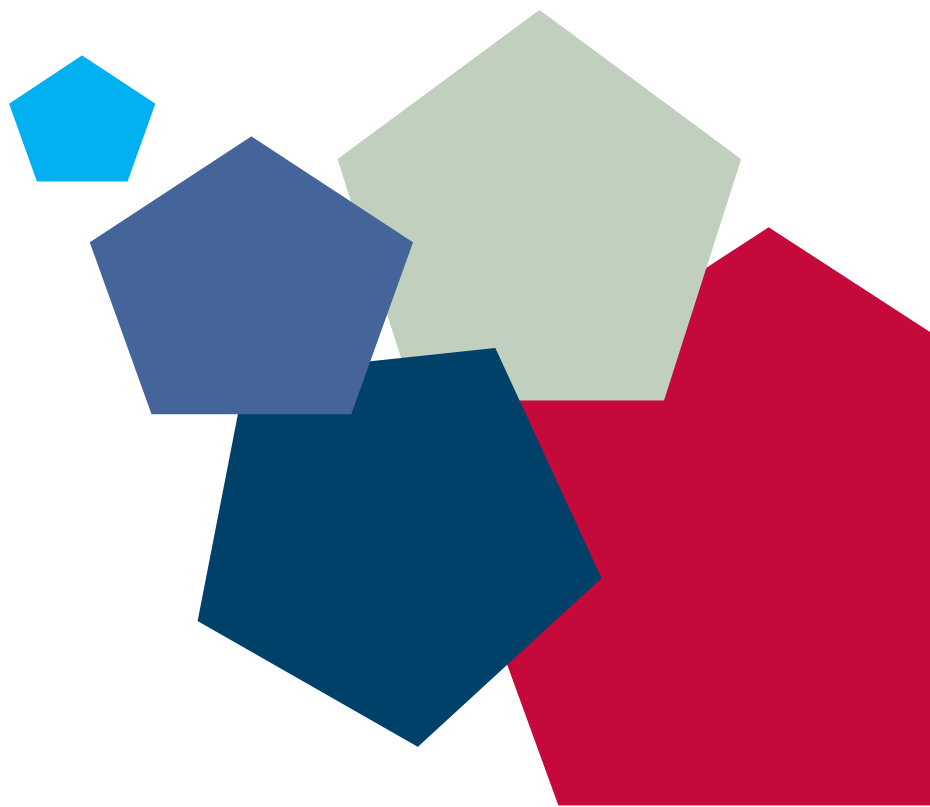


Table of Contents

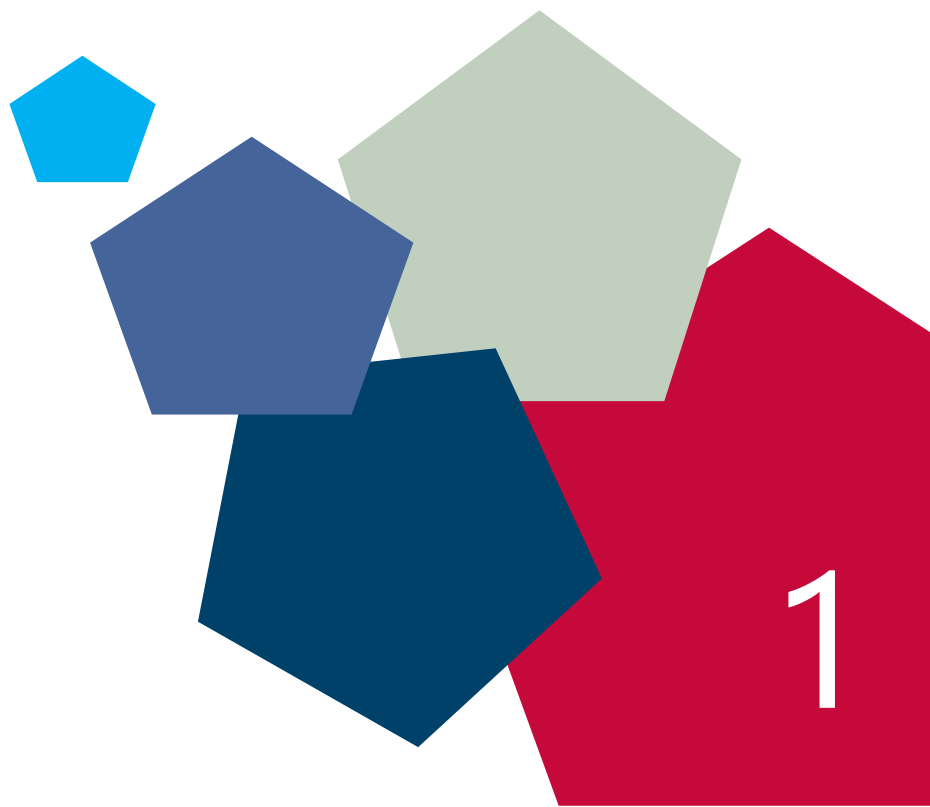
Disclaimer	3
The Challenge – why is it such a pain to work with custom tasks in Biml?.....	4
The Solution – automate your Biml Code by reading from the Task’s DLL.....	6
A real-world example – building a ZIP Task with .NET, Biml and SSIS.....	11
The SSIS Task	11
The Biml Solution	23
Biml Files in Visual Basic (VB.NET).....	24
About the Authors.....	26
Resources	27



Disclaimer

If you have not worked with Biml before, you should look into the basics, first. The idea of this whitepaper is not to provide any primer or basics into Biml.

Nevertheless, if you want to get started using Biml (The Business Intelligence Markup Language), you may want to consider the Resources we've gathered for you at the end of this paper.



The Challenge

– why is it such a pain to work with custom tasks in Biml?

If you've ever worked with custom tasks in SSIS using Biml, you know that their code gets nasty easily. Also, there are parameters like the `CreationName` of the task that you usually would not know but will need to know to actually write the code to access it. In addition, this property can even change over time when installing updates of a specific SSIS task.

Our first example is typical Biml code to build an SSIS package that contains a single task using a custom component. The `CustomTask` tag is what makes it nerve wrecking – the rest is just plain and simple Biml.

E1. Package_with_CustomTask.biml

```
<Biml xmlns="http://schemas.varigence.com/biml.xsd">
  <Packages>
    <Package Name="TestPackage_ReportGenerator"
      ProtectionLevel="EncryptSensitiveWithUserKey">
      <Tasks>
        <CustomTask Name="My SSRS Task"
          CreationName="SSISComponents.ReportGenerator.ReportGenerator,
            SSISComponents.DTS.Tasks.ReportGeneratorTask2017, Version=1.8.5.0, Culture=neutral,
            PublicKeyToken=0ee497b3fc52b917">
          <ObjectData>&lt;InnerObject&gt;
            &lt;Timeout Type="3" Value="30000" /&gt;
            &lt;Reportname Type="8" Value="/Report Project/SampleReport" /&gt;
            &lt;Reportparameter Type="8" Value="" /&gt;
            &lt;Datasetname Type="8" Value="" /&gt;
            &lt;Recordset Type="8" Value="" /&gt;
            &lt;Filename Type="8" Value="C:\temp\sample.pdf"/&gt;
            &lt;ReportServer Type="8" Value="http://localhost/reportserver" /&gt;
          &lt;/InnerObject&gt;
        </CustomTask>
      </Tasks>
    </Package>
  </Packages>
</Biml>
```

```

<!--ReportType Type="8" Value="SQL Server Report" /-->
<!--Username Type="8" Value="" /-->
<!--Domain Type="8" Value="" /-->
<!--Password Type="8" Value="" /-->
<!--PrefixFilename Type="8" Value="" /-->
<!--WindowsAuthorization Type="11" Value="-1" /-->
<!--Prefix Type="11" Value="0" /-->
<!--DebugMode Type="11" Value="0" /-->
<!--Snapshot Type="11" Value="0" /-->
<!--ExecutionDateTime Type="8" Value="" /-->
<!--ExecutionId Type="8" Value="" /-->
<!--ExpirationDateTime Type="8" Value="" /-->
<!--HistoryId Type="8" Value="" /-->
<!--NumPages Type="8" Value="" /-->
<!--TargetServerVersion Type="3" Value="150" /-->
<!--/InnerObject-->
</ObjectData>
</CustomTask>
</Tasks>
</Package>
</Packages>
</Biml>

```

As you can see, the CustomTask requires (besides a name, which is yours to choose) a whole lot of parameters that you cannot control and that can effectively only be derived from the component itself.

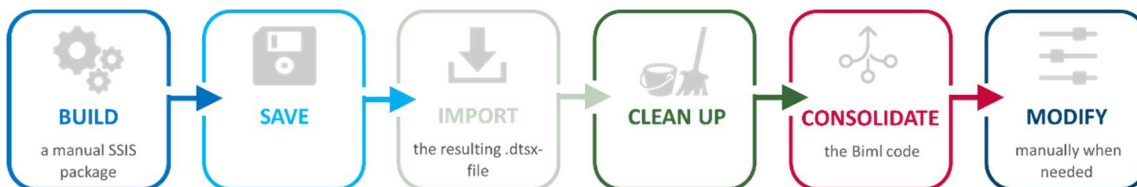


Figure 1: Custom Tasks and Biml – The classic way

The only way to get that done is to build a manual SSIS package using this task, save it, import the resulting .dtsx-file into BimlStudio or BimlExpress, clean up and consolidate the Biml code that you've received from that and manually modify it when needed. This also makes the package very hard to control through metadata which is one of the huge advantages of Biml.

Or is there maybe another, much better way of dealing with this?

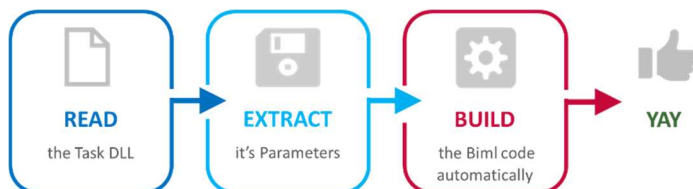


Figure 2: Custom Tasks and Biml – The better way... definitely!



The Solution

– automate your Biml Code by reading from the Task's DLL

As Biml offers full .NET support (using C# or VB.NET), we can use standard .NET functionalities to extract all the required information from the SSIS task's library and therefore automate our Biml code – or automate our automation!

As all SSIS tasks are different but their Biml syntax still follows a joint pattern, it makes sense to split the code into two files: One component specific script and the generic part which can be called using `CallBimlScript`.

Let's look at our calling Biml file (the Caller) first. Each custom task in SSIS requires three properties or attributes.

1. `Name` (the name that will be used for the task within a package)
2. `CreationName` (the component name, version, culture etc.)
3. `ObjectData` (the parameters that are specific to that task)

We'll just store the name of the task in a string and consequently use it for the task itself. The creation name can be extracted from the DLL. Therefore, we need the path to the DLL itself as well as the name of the class that we want to use (as there can potentially be multiple tasks within the same DLL). The path to the DLL is relatively easy, we just have to look up where it got installed. The name of the class may not be as self-explanatory but our solution will provide an answer for that as well.

Finally, we have the component specific parameters which can be anything from 1 to n values. We can concatenate these in a string and separate them

by double pipes ("||"). This leaves us with the flexibility to have as many parameters as we want while keeping the code readable. Feel free to change this to a collection by splitting the string again, which is exactly what we're doing in the Callee. This data should ideally come from the metadata repository that you're using to control your Biml solution.



If you are using Biml but are not using a metadata driven approach, you might want to rethink what you're doing.

E2. GetTask.biml

```
<# String TaskName = @"";
String CompName = @"";
String CompPath = @"";
String Params = @""; #>
<Biml xmlns="http://schemas.varigence.com/biml.xsd">
<Packages>
  <Package Name="TestPackage" ProtectionLevel="EncryptSensitiveWithUserKey">
    <Tasks>
      <#= CallBimlScript("GetTask_CBS.biml",TaskName,CompName,CompPath,Params) #>
    </Tasks>
  </Package>
</Packages>
</Biml>
```

As you can see, all we do is declare the four strings and pass them to the Callee using CallBimlScript.

CompName points to the class to be used, it can remain blank if unknown. Whenever this property is empty, the Callee will not return the Biml code for the component but a list of classes in the assembly that act as SSIS tasks. This can be useful if you do not know the name of the class to be addressed.

So far, this isn't very spectacular. The magic is obviously happening in the Callee...

E3. GetTask_CBS.biml

```
<#@ template designerbimlpath="biml/Packages/Package/Tasks" #>
<#@ property name="TaskName" type="string" #>
<#@ property name="CompName" type="string" #>
<#@ property name="CompPath" type="string" #>
<#@ property name="Params" type="string" #>
<# string TargetServerVersion = @"150"; #>
<# if (CompName == "") { #>
  <!-- Task(s) in Assembly:
  <#= GetCompNames() #>-->
<# }
  else
  {#>
<CustomTask Name="<#= TaskName #>" CreationName="<#= CompName + ", " +
System.Reflection.Assembly.LoadFile(CompPath).FullName #>">
  <ObjectData>&lt;InnerObject&gt;
    <#= InnerObject() #>
    &lt;TargetServerVersion Type="3" Value="<#= TargetServerVersion #>" /&gt;
  &lt;/InnerObject&gt;</ObjectData>
</CustomTask>
<# } #>
<#+
public string GetCompNames() {
  string result = @"";
  foreach (var ComponentProperty in
```



```

System.Reflection.Assembly.LoadFile(CompPath).GetTypes().Where(c => c.GetProperties().Where(d =>
d.PropertyType.ToString() == "Microsoft.SqlServer.Dts.Runtime.DTSTargetServerVersion" ||
d.PropertyType.ToString() == "Microsoft.SqlServer.Dts.ManagedMsg.ErrorSupport").Any())) {
    result = result + ComponentProperty.ToString() + "\n";
}
return result;
}

public string InnerObject() {
    string result = @"";
    foreach (var prop in System.Reflection.Assembly.LoadFile(CompPath).GetTypes().Where(c =>
c.ToString() == CompName).First().GetProperties().Where(d => d.Name != "Version" && d.Name !=
"ExecutionValue" && d.Name != "TargetServerVersion")) {
        result = result + "<" + prop.Name + " Type=\"" +
GetPropertyType(prop.PropertyType) + "\" Value=\"" + GetValue(prop.Name, prop.PropertyType) + "\"
/>\n";
    }
    return result;
}

public string GetValue(string Param, Type pType) {
    string result = @"";
    string ParamName = @"";
    string ParamValue = @"";
    foreach (var pair in Params.Split(new[] { "|" }, StringSplitOptions.None)) {
        ParamName = pair.Split('=')[0];
        ParamValue = "";
        if (ParamName.Length+1 < pair.Length) { ParamValue =
pair.Substring(ParamName.Length+1); }
        if (ParamName.Length > 0 && ParamValue.Length > 0 && ParamName.ToUpper() ==
Param.ToUpper()) { result = ParamValue; }
    }
    if (result == "" && pType.ToString() == "System.Boolean") { result = "0"; }
    return result;
}

public string GetPropertyType(Type pType) {
    string result = @"";
    switch (pType.ToString()) {
        case "System.String":
            result = "8";
            break;
        case "System.Boolean":
            result = "11";
            break;
        case "System.Int32":
            result = "3";
            break;
        default:
            result = "UNTRANSLATED: " + pType.ToString();
            break;
    }
    return result;
}
#>

```



Let us take a closer look at that file. It consists of three main sections:

```
<#@ template designerbimlpath="biml/Packages/Package/Task<br><#@ property name="TaskName" type="string" #><br><#@ property name="CompName" type="string" #><br><#@ property name="CompPath" type="string" #><br><#@ property name="Params" type="string" #><br><# static TargetServerVersion = @"150" #>
```



1. The CallBimlScript header

If you've used CallBimlScript before, there is nothing surprising here. All we do is declare the input parameters as well as one static value (TargetServerVersion). At some point, we'll change that to something dynamic as well.

If you are not familiar with CallBimlScript, we recommend reading Cathrine's blog post on it!

2. The output block

Depending on whether CompName is empty or not, this will either contain the result of the GetCompNames function (included in the code block) or the Biml code for the task. When returning the Biml code, we will get the name for the task from the corresponding variable, build the CreationName attribute from the CompName variable and the FullName of the assembly, add the specific parameters (the InnerObject) from the InnerObject function (see code block) and finish off with the TargetServerVersion property which we'll take from our static value for now (see header).

3. The code block

THIS is finally where the magic happens. Our code block consists of four different functions:

```
<#><br>public string GetCompNames() {<br>    string result = @"";<br>    foreach (var ComponentProperty in<br>        System.Reflection.Assembly.LoadFile(CompPath)<br>        .Properties) {<br>        if (ComponentProperty.PropertyType.ToString() == "Microsoft.SqlSe<br>        d.Property" || ComponentProperty.PropertyType.ToString() == "Microsoft.SqlSe<br>        d.Property") {<br>            result = result + ComponentProperty.Name + "<br>            " + ComponentProperty.PropertyType.ToString() + "<br>            " + ComponentProperty.Value + "<br>            " + Environment.NewLine;<br>        }<br>    }<br>    return result;<br>}
```

3.1 GetCompNames (called within the output block if CompName is empty)

This function will use "**System.Reflection.Assembly.LoadFile**" to open the DLL and return all classes that contain a property of type **Microsoft.SqlServer.Dts.Runtime.DTSTargetServerVersion** or **Microsoft.SqlServer.Dts.ManagedMsg.ErrorSupport**.

The presence of either property is a pretty good sign that this is an SSIS task.

As we will only use this information within a comment to display the potential classes within the DLL, we can just call the ToString method and line separate them.

```
public string InnerObject() {<br>    string result = @"";<br>    foreach (var prop in System.Reflection.<br>        Assembly.LoadFile(CompPath).Properties) {<br>        if (prop.PropertyType.ToString() == "Microsoft.SqlServer.Dts.ManagedMsg.<br>        ErrorSupport" || prop.PropertyType.ToString() == "Microsoft.SqlServer.Dts.<br>        DTSTargetServerVersion") {<br>            result = result + prop.Name + "<br>            " + prop.Value + "<br>            " + Environment.NewLine;<br>        }<br>    }<br>    return result;<br>}
```

3.2 InnerObject (called within the Biml output if CompName is not empty)

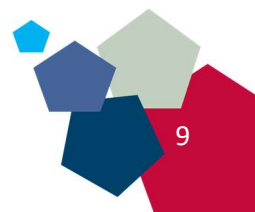
Just like the first function, this one will load the DLL but will look specifically at the class provided using the CompName property.

For each property it will create a line in encoded XML that consists of the name of that property, its data type (using GetPropertyType) and its value (using GetValue).

```
public string GetPropertyType(Type pType) {<br>    string result = @"";<br>    switch (pType.ToString()) {<br>        case "System.String":<br>            result = "8";<br>            break;<br>        case "System.Int32":<br>            result = "4";<br>            break;<br>        case "System.Boolean":<br>            result = "1";<br>            break;<br>        default:<br>            result = "0";<br>            break;<br>    }<br>    return result;<br>}
```

3.3 GetPropertyType (called by InnerObject)

SSIS uses an internal ENUM for data types within SSIS custom tasks. This function will simply translate the data type that was passed to it and return the corresponding ENUM value for it. You may potentially have to adjust this function when calling a DLL that uses data types different from String, Int32 or Boolean.



```

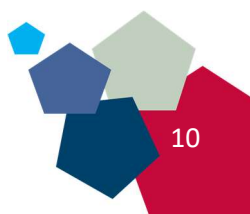
public string GetValue(string Param, Type pType)
// to be done
string result = "";
string ParamName = "";
string ParamValue = "";
// Search for value in Param string

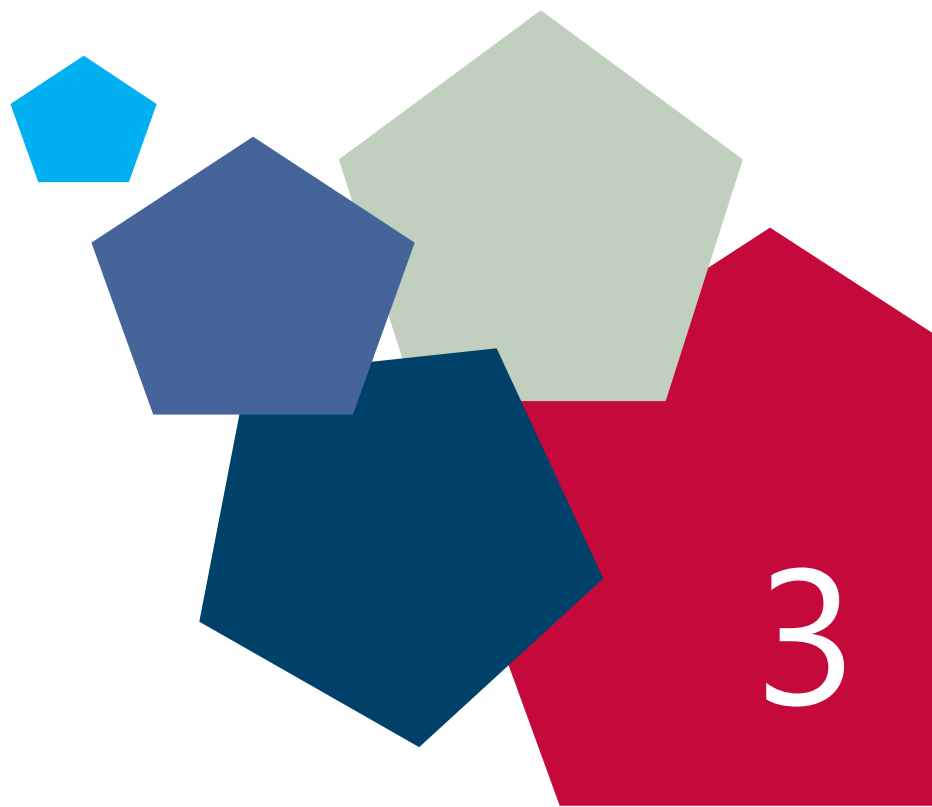
```

3.4 GetValue (called by InnerObject)

This function will split the Params string into single parameters (splitting by "|"") and then split each parameter into a name and a value (splitting by "="). It will then check if a value was provided for the current parameter and if so, return this value.

If no value was provided for this parameter and the data type is Boolean, it will return 0.





A real-world example – building a ZIP Task with .NET, Biml and SSIS

The SSIS Task

If you want to extend the SQL Server Integration Services with your own functions or logic, you must first ask yourself the basic question whether you want to develop an SSIS component or an SSIS task.

An SSIS component extends the data flow with additional functions. Here it should be additionally differentiated whether a data source, a transformation or a destination is to be developed. An SSIS Task extends the Control Flow by additional functions.

In this chapter we will use the example of developing an SSIS ZIP Task. The development of an SSIS component is basically similar but much more complex due to the entire handling of the metadata and would therefore go beyond the scope of the actual topic of this white paper.



The ZIP task to be developed should be able to pack and unpack files. When packing files, additional parameters such as the ZIP level, a password and the AES key size should be specified.

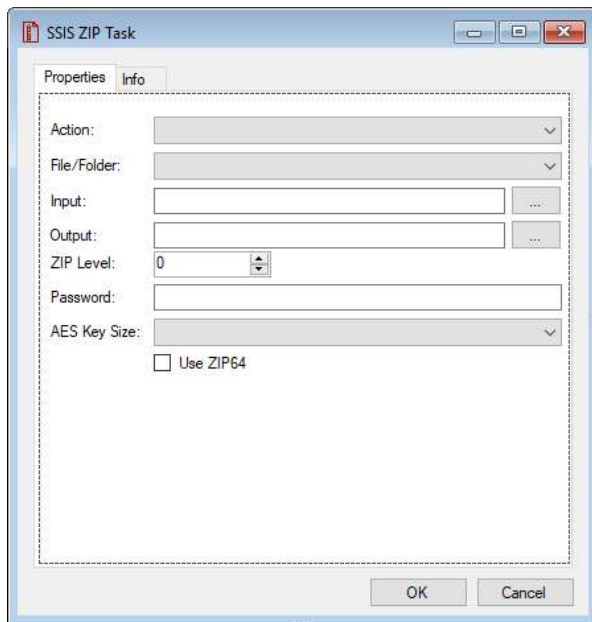


Figure 3: SSIS ZIP Task - UI

For our ZIP Task we rely on the open source library SharpZipLib.



The SharpZipLib is a compression library for Zip, Gzip, Bzip2 and Tar files. Further information about SharpZipLib can be found on the project page icsharpcode.github.io/SharpZipLib

Installation folders

Before we start with the actual development of the SSIS Task, we must take a short excursion into the Global Assembly Cache and the installation directories of an SSIS Task. Each task must exist in 2 different directories.

The first is the directory `%PROGRAMFILES(X86)%\Microsoft SQL Server\NNN\DTS\Tasks` where the directory `NNN` represents the version number of the SQL Server for which an SSIS package is to be developed.



The version numbers in the directories correspond to the following names of the SQL Server versions:

110 = SQL Server 2012 = 11.00.xxxx

120 = SQL Server 2014 = 12.00.xxxx

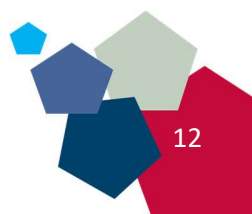
130 = SQL Server 2016 = 13.00.xxxx

140 = SQL Server 2017 = 14.00.xxxx

150 = SQL Server 2019 = 15.00.xxxx

Since version 2015 of SQL Server Data Tools, the version of the target server can easily be changed when developing an SSIS package via the properties of a solution. For these changes to be supported, the task must be stored in the respective directory for each version.

The SQL Server Developer Studio (SSDT) accesses the tasks in this directory during the development phase. If we were to develop another extension instead of a task, this would be stored in one of the other directories instead of in the Task directory.



<input type="checkbox"/> Name	Date modified	Type	Size
Binn	19.12.2018 15:44	File folder	
Connections	02.12.2018 21:06	File folder	
DataDumps	30.10.2018 13:38	File folder	
Extensions	02.12.2018 21:06	File folder	
ForEachEnumerators	19.12.2018 15:42	File folder	
LogProviders	30.10.2018 13:38	File folder	
MappingFiles	30.10.2018 13:38	File folder	
Packages	30.10.2018 13:38	File folder	
PipelineComponents	19.12.2018 15:44	File folder	
ProviderDescriptors	30.10.2018 13:38	File folder	
Tasks	19.12.2018 15:44	File folder	
UpgradeMappings	30.10.2018 13:39	File folder	

Figure 4: SSDT - Folder Structure

The second directory in which each task and component must be installed is the Global Assembly Cache (GAC). The GAC was introduced with the .NET Framework to avoid version conflicts between program libraries. However, a task cannot simply be copied into the directory but must be installed into the GAC using `Gacutil.exe`.



`Gacutil.exe` is automatically installed with Visual Studio. Further information about the `Gacutil` can be found in the Microsoft Docs.

To install an assembly into the GAC, it must have a strong name, giving it a unique identity. Assigning a strong name to a task or component during development is relatively easy and will be explained later.

To be able to use third-party assemblies in your own SSIS tasks, they must also have a strong name so that they can be installed in the GAC. Every component that is used in the project and is not part of the .NET Framework must be present in the GAC as well as in the SQL Server directory.

The used `SharpZipLib` can fortunately be obtained directly from Nuget with a strong name. If a component to be used does not have a strong name, it must be recompiled with a strong name. If this is not possible, an existing assembly can alternatively be provided with a strong name afterwards. Have a look at the information about the tools `sn.exe` as well as `ildasm.exe` and `ilasm.exe` in the Microsoft Docs.



The Global Assembly Cache can be found in the directory `C:\Windows\Microsoft.NET\assembly`



Once the component has been compiled, it must be placed in the directory `%PROGRAMFILES(X86)%\Microsoft SQL Server\NNN\DTS\Tasks`

and be installed in the Global Assembly Cache with:

```
gacutil.exe -i ICSharpCode.SharpZipLib.dll
gacutil.exe -i oh22is.DTS.Task.ZIP2017.dll.
```

Assemblies can be uninstalled with the command:
`gacutil.exe -u oh22is.DTS.Task.ZIP2017`
from the GAC again.

Development of the SSIS Task

To develop an SSIS component, a new Class Library (.NET Framework) project must be created.

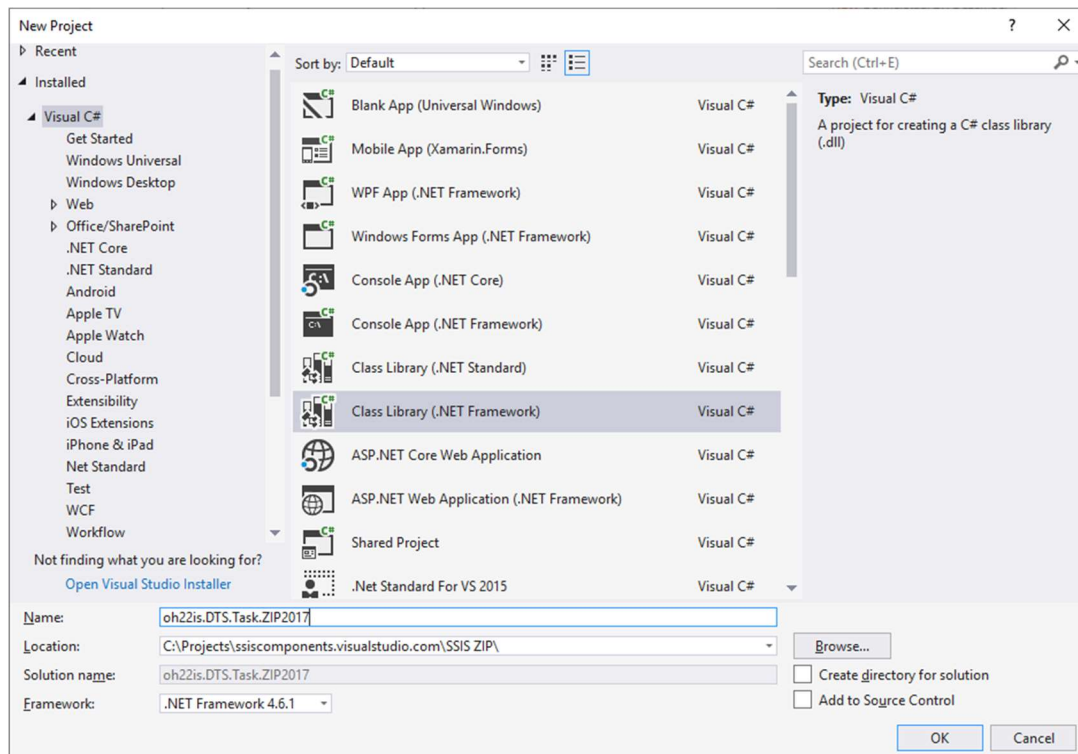


Figure 5: SSDT – New Project – Class Library

Then references to the assemblies **Microsoft.SqlServer.Dts.Design** and **Microsoft.SqlServer.ManagedDTS** must be added to the project. These assemblies are required to implement the appropriate interfaces for the SSIS task. Since we also want to provide the SSIS Task with a user interface but have selected Class Library as the project type, a reference to **System.Windows.Forms** must also be added.

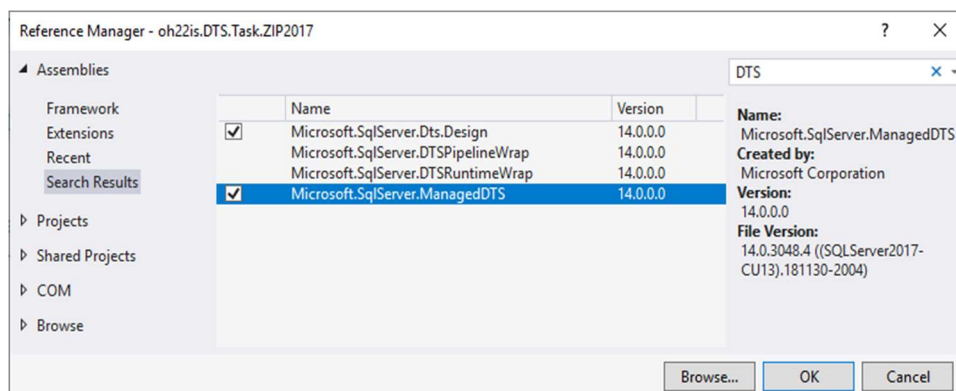


Figure 6: Reference Manager – Add Project References

For our exemplary SSIS task you have to add the SharpZipLib package via NuGet.

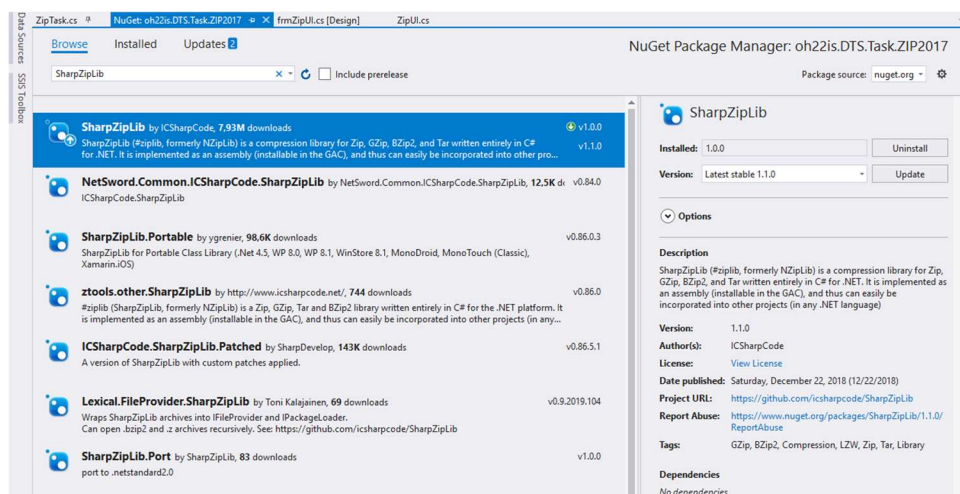


Figure 7: ZipTask – NuGet Package Manager – Add NuGet Packages

To give the task a strong name, the assembly must be signed in the solution properties. By checking 'Sign the assembly' a filename for the strong name key file is defined. It is not necessary to encrypt the file with a password.

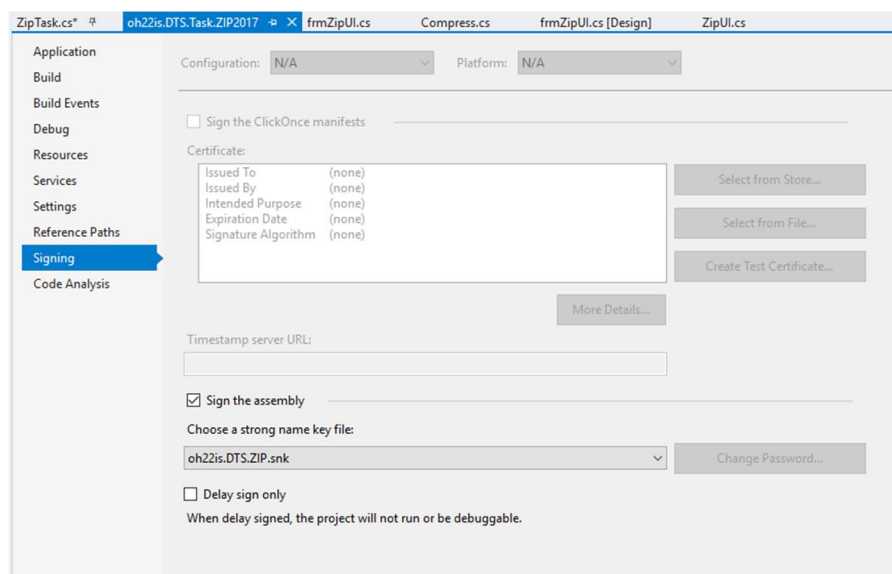


Figure 8: ZipTask – Signing – Sign the assembly

After all references for the SSIS Task have been added, we need to implement the two essential interfaces for an SSIS Task in our project.

Microsoft.SqlServer.Dts.Runtime.Task

In the first step, we have to implement the abstract class Task. Therefore, we create the class ZipTask in our project. The class gets a using statement on **Microsoft.SqlServer.Dts.Runtime**.

Now different attributes have to be added to the class. The two essential attributes are:

DisplayName - The name of the task as should be displayed in the SSIS Toolbox

UITypeName - A property to associate the task with the individual user interface

Other attributes that can be added are among others:

IconResource - Providing a specific icon for the task

TaskType - Meta information to place the task in the SSIS Toolbox

TaskContact - Contact information for the end user

RequiredProductLevel - SQL Server Edition (Standard, Enterprise, None)



The interface `Microsoft.SqlServer.Dts.Runtime.Task` must then be implemented by the class. A task has several methods of which we only want to overwrite the `Execute` method. The `Execute` method is the method that performs the actual work when the package is executed, in this case compressing or decompressing.

Within the class, the public variables that we want to use to configure the component and control its execution are defined.

E4. Class ZipTask

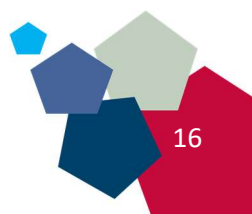
```
using System;
using Microsoft.SqlServer.Dts.Runtime;

namespace oh22is.DTS.Task.ZIP
{
    [DtsTask(DisplayName = "SharpZipLib Task",
        Description = "ZIP Task for SSIS using SharpZipLib. SharpZipLib (#ziplib, formerly NZipLib) is a compression library that supports Zip files using both stored and deflate compression methods, PKZIP 2.0 style and AES encryption, tar with GNU long filename extensions, GZip, zlib and raw deflate, as well as BZip2.",
        IconResource = "oh22is.DTS.Task.ZIP.zip.ico",
        UITypeName = "oh22is.DTS.Task.ZIP.ZipUI, oh22is.DTS.Task.ZIP2017, Version=1.0.0.0, Culture=Neutral, PublicKeyToken=28fb6a90a3947f07",
        TaskContact = "Tillmann Eitelberg, oh22information services GmbH",
        RequiredProductLevel = DTSPProductLevel.None)]
    public class ZipTask : Microsoft.SqlServer.Dts.Runtime.Task
    {
        /// <summary>
        /// Variables / SSIS Properties for the configuration of the component
        /// </summary>
        #region Properties

        public string Action { get; set; }
        public string FileFolder { get; set; }
        public string Input { get; set; }
        public string Output { get; set; }
        public decimal ZipLevel { get; set; }
        public string Password { get; set; }
        public int AesKeySize { get; set; }
        public bool Zip64 { get; set; }

        #endregion Properties

        /// <summary>
        ///
    }
```




```

    /// </summary>
    /// <param name="connections"></param>
    /// <param name="variableDispenser"></param>
    /// <param name="componentEvents"></param>
    /// <param name="log"></param>
    /// <param name="transaction"></param>
    /// <returns></returns>
    public override DTSExecResult Execute(Connections connections, VariableDispenser
variableDispenser, IDTSComponentEvents componentEvents, IDTSLogging log, object transaction)
    {
        bool fireAgain = false;
        try
        {
            if (Action == "ZIP")
            {
                ...
                ...
                ...
                ...
                componentEvents.FireInformation(0, "SSIS ZIP Task", Input + " has been
compressed", null, 0, ref fireAgain);
                return DTSExecResult.Success;
            }
            else
            {
                ...
                ...
                ...
                ...
                componentEvents.FireInformation(0, "SSIS ZIP Task", Output + " has been
uncompressed", null, 0, ref fireAgain);
                return DTSExecResult.Success;
            }
        }
        catch (Exception ex)
        {
            componentEvents.FireError(0, "SSIS ZIP Task", ex.ToString(), null, 0);
            return DTSExecResult.Failure;
        }
    }
}

```

The actual code for compressing/decompressing has been removed from the example code for clarity. The complete code can be found on [GitHub](#).



Within the Execute method `FireInformation` and `FireError` are called. These calls can be used to write various notifications from the component to the SSIS log.

Another possibility here would be to output the status of the compression/decompression process via `FireProgress`. However, it should be noted that the firing of events costs a lot of time and can therefore affect the performance of the packages.

Another method that could be implemented within our class is the `Validate` method. This method is called at several points during the development and execution of an SSIS package and is used to verify that the component has been properly configured. In our example, this method is currently not overwritten.



UITypeName

The `UITypeName` is a property to associate the task with the individual user interface.

The format of the property is a comma separated string containing the following values:

`oh22is.DTS.Task.ZIP.ZipUI`

TypeName: `oh22is.DTS.Task.ZIP.ZipUI` is the full class name including namespace of the user interface; see next section for more information.

`oh22is.DTS.Task.ZIP2017`

Assembly Name: `oh22is.DTS.Task.ZIP2017` is the name of the assembly. Since the UI in our component is part of the actual component, we refer here to "itself". In principle, it is also possible to outsource the user interface to another assembly.

`Version=1.0.0.0`

File Version: `Version=1.0.0.0` defines the version of the component.

`Culture=Neutral`

Culture: The culture of the component

`PublicKeyToken=28fb6a90a3947f07`

Public key token: The public key token of the generated assembly. To read this value, the component must be built once using Visual Studio. Then the public key token can be read with the tool `sn.exe`. This program is located in the installation directory of Visual Studio.

Via `sn -T oh22is.DTS.Task.ZIP2017` the Public Key Token can be read.


<input type="checkbox"/> Name	Date modified	Type	Size
 v4.0_1.0.0.0_28fb6a90a3947f07	06.01.2019 18:56	File folder	

Figure 9: Global Assembly Cache

The corresponding key is also displayed as part of the assembly in the GAC as soon as it has been installed in the GAC via `gacutil.exe`.

IconResource

If you develop your own component, you might also like to assign your own icon to it. For this you have to add your own ICO file to the project. In our case the file is called `zip.ico`. The property Build Action of this file must be set to Embedded Resource. The value within the class attribute then consists of the complete default namespace of the project (`oh22is.DTS.Task.ZIP`) as well as the file name `zip.ico`.

Since we want to give our component an interface for configuration, we now create the class ZipUI in our project. The following using statements must be added to this class:

```
using System;
using System.Windows.Forms;
using Microsoft.SqlServer.Dts.Runtime;
using Microsoft.SqlServer.Dts.Runtime.Design;
```

In order for the UI to be able to work with the component and the connections within the package, we add the two private variables `_taskHost` and `_connections` to the class. These are initialized during the initialization of the class.

The interface expects the methods `Delete`, `GetView`, `Initialize` and `New` to be implemented. In our task the two methods `Delete` and `New` will not be executing any code.

E5. Class ZipUI

```
using System;
using System.Windows.Forms;
using Microsoft.SqlServer.Dts.Runtime;
using Microsoft.SqlServer.Dts.Runtime.Design;

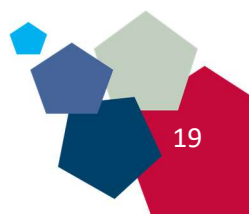
namespace oh22is.DTS.Task.ZIP
{
    class ZipUI : Microsoft.SqlServer.Dts.Runtime.Design.IDtsTaskUI
    {
        private TaskHost _taskHost;
        private Connections _connections;

        public void Delete(IWin32Window parentWindow)
        {
        }

        public ContainerControl GetView()
        {
            // Launch your editor with information from your task and available connections
            return new frmZipUI(_taskHost, _connections);
        }

        public void Initialize(TaskHost taskHost, IServiceProvider serviceProvider)
        {
            _taskHost = taskHost;
            var cs = serviceProvider.GetService(typeof(IDtsConnectionService)) as
IDtsConnectionService;
            if (cs != null) _connections = cs.GetConnections();
        }

        public void New(IWin32Window parentWindow)
        {
        }
    }
}
```



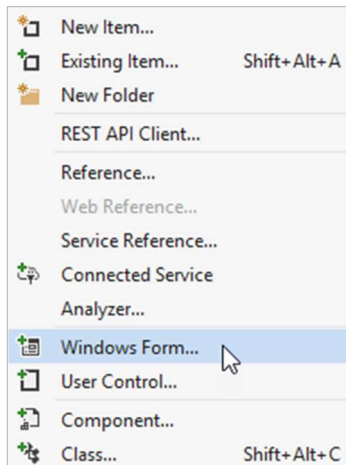


Figure 10: Add Windows Form

In order to add an interface to our component, another item must be added to our project. This time however no class but a Windows Form Item with the name frmZipUi.



Our component can be configured by the user using this Windows Form UI. The Windows Form also requires a using statement on `Microsoft.SqlServer.Dts.Runtime`.

Besides the UI methods only 2 lines of code are required for our component. One of them is the assignment of values from the properties to the UI:

```
cbAction.SelectedItem = _taskHost.Properties["Action"].GetValue(_taskHost) == null ?
"ZIP" : _taskHost.Properties["Action"].GetValue(_taskHost).ToString();
```

and the other the assignment of new values from the UI to the properties:

```
_taskHost.Properties["Action"].SetValue(_taskHost, cbAction.SelectedItem.ToString());
```

Be sure to test the properties for NULL before using them. If you access properties with NULL values, an exception is thrown. The calls are executed when the component is opened and when the component is closed via the OK button respectively.

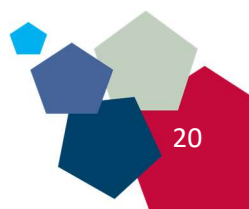
E6. Class frmZipUi

```
using System;
using System.Windows.Forms;
using Microsoft.SqlServer.Dts.Runtime;

namespace oh22is.DTS.Task.ZIP
{
    public partial class frmZipUI : Form
    {
        TaskHost _taskHost;
        Connections _connections;

        public frmZipUI(TaskHost taskHost, Connections connections)
        {
            InitializeComponent();
            _taskHost = taskHost;
            _connections = connections;
        }

        private void frmZipUI_Load(object sender, EventArgs e)
        {
            cbAction.SelectedItem = _taskHost.Properties["Action"].GetValue(_taskHost) == null ?
            "ZIP" : _taskHost.Properties["Action"].GetValue(_taskHost).ToString();
            cbFileFolder.SelectedItem = _taskHost.Properties["FileFolder"].GetValue(_taskHost) ==
            null ? "FILE" : _taskHost.Properties["FileFolder"].GetValue(_taskHost).ToString();
            txtInput.Text = _taskHost.Properties["Input"].GetValue(_taskHost) == null ? "" :
            _taskHost.Properties["Input"].GetValue(_taskHost).ToString();
        }
    }
}
```



```

        txtOutput.Text = _taskHost.Properties["Output"].GetValue(_taskHost) == null ? "" :
        _taskHost.Properties["Output"].GetValue(_taskHost).ToString();
        nudZipLevel.Value = _taskHost.Properties["ZipLevel"].GetValue(_taskHost) == null ? 0 :
        Convert.ToDecimal(_taskHost.Properties["ZipLevel"].GetValue(_taskHost));
        txtPassword.Text = _taskHost.Properties["Password"].GetValue(_taskHost) == null ? "" :
        _taskHost.Properties["Password"].GetValue(_taskHost).ToString();
        cbAESKeySize.SelectedItem = _taskHost.Properties["AesKeySize"].GetValue(_taskHost) ==
        null ? 0 : Convert.ToDecimal(_taskHost.Properties["AesKeySize"].GetValue(_taskHost));
        cbZip64.Checked = _taskHost.Properties["Zip64"].GetValue(_taskHost) == null ? true :
        Convert.ToBoolean(_taskHost.Properties["Zip64"].GetValue(_taskHost));
    }

    private void btnOK_Click(object sender, EventArgs e)
    {
        _taskHost.Properties["Action"].SetValue(_taskHost, cbAction.SelectedItem.ToString());
        _taskHost.Properties["FileFolder"].SetValue(_taskHost,
        cbFileFolder.SelectedItem.ToString());
        _taskHost.Properties["Input"].SetValue(_taskHost, txtInput.Text);
        _taskHost.Properties["Output"].SetValue(_taskHost, txtOutput.Text);
        _taskHost.Properties["ZipLevel"].SetValue(_taskHost, nudZipLevel.Value);
        _taskHost.Properties["Password"].SetValue(_taskHost, txtPassword.Text);
        _taskHost.Properties["AesKeySize"].SetValue(_taskHost,
        cbAESKeySize.SelectedItem.ToString());
        _taskHost.Properties["Zip64"].SetValue(_taskHost, cbZip64.Checked);
        DialogResult = DialogResult.OK;
    }
}
...
...
...
...
...
}

```

Debugging

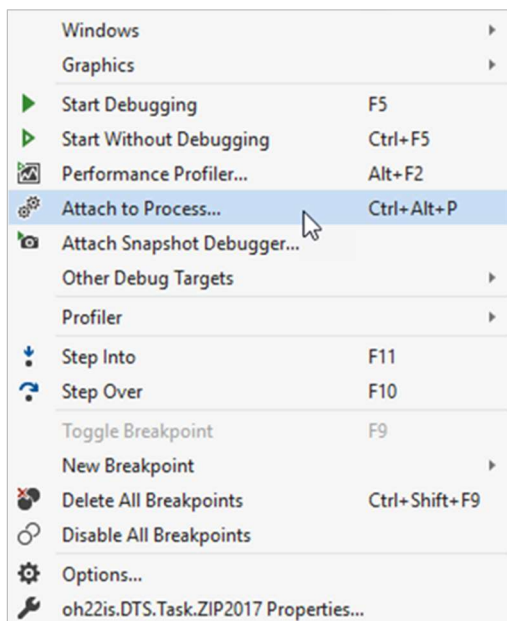


Figure 11: Attach to Process – Context menu

In order to debug an SSIS component during development, you need to attach to a Visual Studio in which an SSIS package is being developed. Make sure that you copy the assemblies after each build process into the respective SQL Server directories and install them in the GAC.

The process to connect to is called `devenv.exe`. If several instances of Visual Studio are open, you need to make sure that the name of the corresponding SSIS solution is displayed in the Title column.

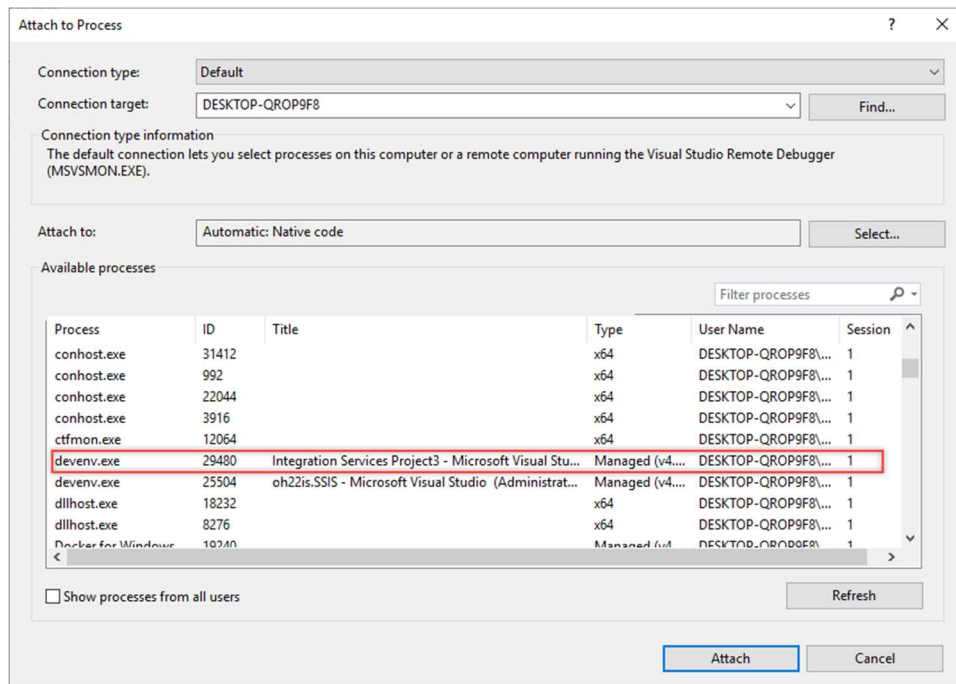


Figure 12: Attach to Process – Process explorer

To simplify this procedure, an installation script can be defined in the project properties under Build Events, which automatically copies the corresponding assemblies into the respective SQL Server folders after the build process and installs the component in the GAC. Under Debug Visual Studio can be defined as an external program to be started as a Start Action. This way the debugging in a new Visual Studio process can take place directly via the integrated "Start Debugging" process.

The Biml Solution

Now we can pass the required parameters to the called Biml script (the Callee).

For the ZIP component, we need the generic parameters:

Parameter	Value
TaskName	My ZIP Task
CompPath	C:\MyZIP\oh22is.DTS.Task.ZIP2017.dll
CompName	oh22is.DTS.Task.ZIP.ZipTask

Of course, these may differ depending on your component and configuration. In addition, we will pass the component specific parameters in one concatenated string:

Parameter	Value
Action	ZIP
FileFolder	FOLDER
Input	C:\SampleZIPFolder
Output	C:\Temp\sample.zip

Our resulting Biml file looks like this:

E7. GetTask_ZIP.biml

```
<# String TaskName = @"My ZIP Task";
String CompPath = @"C:\MyZIP\oh22is.DTS.Task.ZIP2017.dll";
String CompName = @"oh22is.DTS.Task.ZIP.ZipTask";
String Params =
@"Action=ZIP|FileFolder=FOLDER|Input=C:\SampleZIPFolder|Output=C:\Temp\sample.zip"; #>

<Biml xmlns="http://schemas.varigence.com/biml.xsd">
  <Packages>
    <Package Name="TestPackage_ZIP" ProtectionLevel="EncryptSensitiveWithUserKey">
      <Tasks>
        <#= CallBimlScript("GetTask_CBS.biml",TaskName,CompName,CompPath,Params) #>
      </Tasks>
    </Package>
  </Packages>
</Biml>
```

The other file, GetTask_CBS.biml, remains unchanged, as it is completely generic for all kinds of SSIS Task DLLs.

Appendix I: Biml Files in Visual Basic (VB.NET)

For your reference, we're also providing the Biml code using Visual Basic (VB.NET) instead of C#.

E8. GetTask_ZIP_VB.biml

```
<#@ template language="VB" optionexplicit="False" #>
<# TaskName = "My ZIP Task"
    CompPath = "C:\Temp\oh22is.DTS.Task.ZIP2017.dll"
    CompName = "oh22is.DTS.Task.ZIP.ZipTask"
    Params =
"Action=ZIP||FileFolder=FOLDER||Input=C:\SampleZIPFolder||Output=C:\Temp\sample.zip" #>
<Biml xmlns="http://schemas.varigence.com/biml.xsd">
<Packages>
    <Package Name="TestPackage_ZIP" ProtectionLevel="EncryptSensitiveWithUserKey">
        <Tasks>
            <#= CallBimlScript("GetTask_VB_CBS.biml",TaskName,CompName,CompPath,Params) #>
        </Tasks>
    </Package>
</Packages>
</Biml>
```

E9. GetTask_VB_CBS.biml

```
<#@ template language="VB" optionexplicit="False" designerbimlpath="biml/Packages/Package/Tasks"
#>
<#@ property name="TaskName" type="string" #>
<#@ property name="CompName" type="string" #>
<#@ property name="CompPath" type="string" #>
<#@ property name="Params" type="string" #>
<# TargetServerVersion = "150" #>
<# if CompName = "" then #>
<!-- Task(s) in Assembly:
<#= GetCompNames() #>-->
<# else #>
```



```

<CustomTask Name="<#= TaskName #>" CreationName="<#= CompName & ", " &
System.Reflection.Assembly.loadfile(CompPath).FullName #>">
    <ObjectData>&lt;InnerObject&gt;
        <#= InnerObject #>
        &lt;TargetServerVersion Type="3" Value="<#= TargetServerVersion #>" /&gt;
        &lt;/InnerObject&gt;</ObjectData>

</CustomTask>
<# end if #>

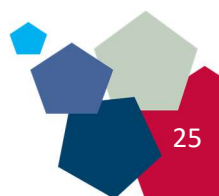
<#+ Function InnerObject() as string
    For Each prop In
System.Reflection.Assembly.loadfile(CompPath).GetTypes().Where(Function(c) c.ToString() =
CompName).First.GetProperties().Where(Function(c) c.Name <> "Version" And c.Name <>
"ExecutionValue" And c.Name <> "TargetServerVersion")
        InnerObject = InnerObject & "&lt;" & prop.Name & " Type=" & " &
GetPropertyType(prop.PropertyType) & " Value=" & GetValue(prop.Name,prop.PropertyType) & "
/&gt;" & vbCrLf
    Next
end function

    Function GetCompNames() as string
        for Each ComponentProperty In
System.Reflection.Assembly.loadfile(CompPath).GetTypes().Where(function(c)
c.GetProperties().Where(function(d) d.PropertyType.ToString =
"Microsoft.SqlServer.Dts.Runtime.DTSTargetServerVersion" or d.propertytype.toString =
"Microsoft.SqlServer.Dts.ManagedMsg.ErrorSupport").Any)
            GetCompNames = GetCompNames & ComponentProperty.ToString() & vbCrLf
        next
    end function

    Function GetValue(Param As String, ptype as Type) As String
ParamValueOutput = ""
        for each pair in Params.Split("||")
            ParamName = pair.split("=")(0)
            ParamValue = ""
            If ParamName.Length+1 < pair.length then ParamValue =
pair.SubString(ParamName.Length+1)
            If ParamName.length > 0 and ParamValue.Length > 0 and ParamName.ToUpper =
Param.ToUpper then ParamValueOutput =ParamValue
        next
        if ParamValueOutput = "" and ptype.ToString = "System.Boolean" then ParamValueOutput ="0"
return ParamValueOutput
    End Function

    Function GetPropertyType(pType As Type) As String
Select Case ptype.ToString
    Case "System.String"
        GetPropertyType = "8"
    Case "System.Boolean"
        GetPropertyType = "11"
    Case "System.Int32"
        GetPropertyType = "3"
    Case Else
        GetPropertyType = "UNTRANSLATED: " & ptype.ToString()
End Select
End Function #>

```



Appendix II

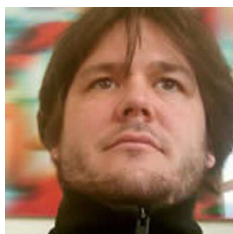
About the Authors

Ben Weissman



Ben is the owner and founder of Solisyon, a consulting firm based in Germany and focused on business intelligence, business analytics, and data warehousing. He is a Microsoft Data Platform MVP, the first German BimlHero, and has been working with SQL Server since SQL Server 6.5. Ben is also an MCSE, Charter Member of the Microsoft Professional Program for Big Data, Artificial Intelligence and Data Science, and he is a Certified Data Vault Data Modeler. If he's not currently working with data, he is probably travelling to explore the world. You can find him online at @bweissman on Twitter, he blogs at <http://biml-blog.de>.

Tillmann Eitelberg



Tillmann is an owner and co-founder of oh22information services GmbH, a consulting company based in Germany and focused on Azure data technologies, data movement and data quality. He has published several open source projects, which were previously published on Codeplex and partly now on GitHub. He has also worked with Microsoft to develop open source SSIS components for SQL Server Data Quality Services. He is a Microsoft Data Platform MVP, PASS Rheinland Chapter Leader, vice-president of PASS Germany e.V. and a Regional Mentor for PASS Global.

Resources



Books

1. The Biml Book, ISBN: 9781484231340



Blogs

2. sqlservercentral.com/stairway/100550/
3. solisyon.de/metadata-based-ssis-biml-1/
4. cathrinewilhelmsen.net/2015/02/23/dont-repeat-your-biml-callbimlscript/
5. Biml-Blog.de



The Internet

6. bimlscript.com
7. icsharpcode.net
8. docs.microsoft.com/en-us/dotnet/framework/tools/gacutil-exe-gac-tool
9. docs.microsoft.com/en-us/dotnet/framework/app-domains/strong-named-assemblies
10. nuget.org
11. docs.microsoft.com/en-us/dotnet/framework/tools/sn-exe-strong-name-tool
12. docs.microsoft.com/en-us/dotnet/framework/tools/ildasm-exe-il-disassembler
13. docs.microsoft.com/en-us/dotnet/framework/tools/ilasm-exe-il-assembler
14. docs.microsoft.com/en-us/dotnet/api/microsoft.sqlserver.dts.runtime.dtstaskattribute?view=sqlserver-2017
15. docs.microsoft.com/en-us/dotnet/api/microsoft.sqlserver.dts.runtime.task?view=sqlserver-2017
16. docs.microsoft.com/en-us/dotnet/api/microsoft.sqlserver.dts.runtime.idtscomponentevents?view=sqlserver-2017
17. docs.microsoft.com/de-de/dotnet/framework/tools/gacutil-exe-gac-tool



Downloads

18. github.com/icsharpcode/SharpZipLib
19. icsharpcode.github.io/SharpZipLib
20. www.nuget.org/packages/SharpZipLib
21. github.com/brutaldev/StrongNameSigner
22. github.com/solisyon/BimlCustomSSISTasks

Illustrations

Figure 1: Custom Tasks and Biml – The classic way	5
Figure 2: Custom Tasks and Biml – The better way... definitely!	5
Figure 3: SSIS ZIP Task - UI.....	12
Figure 4: SSDT - Folder Structure	13
Figure 5: SSDT – New Project – Class Library	14
Figure 6: Reference Manager – Add Project References	14
Figure 7: ZipTask – NuGet Package Manager – Add Nuget Packages	15
Figure 8: ZipTask – Signing – Sign the assembly.....	15
Figure 9: Global Assembly Cache	18
Figure 10: Add Windows Form.....	20
Figure 11: Attach to Process – Context menu	21
Figure 12: Attach to Process – Process explorer	22

Coding Examples

E1. Package_with_CustomTask.biml.....	4
E2. GetTask.biml.....	7
E3. GetTask_CBS.biml.....	7
E4. Class ZipTask.....	16
E5. Class ZipUI.....	19
E6. Class frmZipUi.....	20
E7. GetTask_ZIP.biml.....	23
E8. GetTask_ZIP_VB.biml.....	24
E9. GetTask_VB_CBS.biml.....	24

