A. Leonard, *Building Custom Tasks for SQL Server Integration Services*
https://doi.org/10.1007/978-1-4842-6482-9_18

18. Instrumentation and Validation

Andy Leonard[1]
(1) Farmville, VA, USA

Instrumentation is a sub-science of engineering which originally meant collecting physical data from industrial processes. In software engineering, instrumentation refers to the art and science of surfacing key indicators of the settings, state, and performance of software.

Validation is checking values to make certain the values are in a usable state. In the context of the Execute Catalog Package Task, certain properties *must* be configured using non-empty and non-default values.

In this chapter, we will add instrumentation and validation to the Execute Catalog Package Task.

Instrumentation

In its most basic form, instrumentation is how code "talks" to developers and operators – it informs developers and operators of the status of an operation. In the case of the ExecuteCatalogPackageTask, it is expected that the task inform developers and operators if the SSIS Catalog package executed successfully or not.

Adding Instrumentation

Open the ExecuteCatalogPackageTask.cs file in the ExecuteCatalogPackageTask project in the ExecuteCatalogPackageTask solution, and create a new function named `logMessage` in the `ExecuteCatalogPackageTask` class using the code in Listing 18-1:

```
private void logMessage(
  IDTSComponentEvents componentEvents
, string messageType
, int messageCode
, string subComponent
```

```
, string message)
{
  bool fireAgain = true;
  switch(messageType)
  {
    default:
      break;
    case "Information":
      componentEvents.FireInformation(messageCode, subComponent, message
      break;
    case "Warning":
      componentEvents.FireWarning(messageCode, subComponent, message, ""
      break;
    case "Error":
      componentEvents.FireError(messageCode, subComponent, message, "",
      break;
  }
}
```

Listing 18-1 Adding logMessage

Once added, the code appears as shown in Figure 18-1:

```
private void logMessage(IDTSComponentEvents componentEvents
    , string messageType
    , int messageCode
    , string subComponent
    , string message)
{
    bool fireAgain = true;
    switch(messageType)
    {
        default:
            break;
        case "Information":
            componentEvents.FireInformation(messageCode, subComponent, message, "", 0, ref fireAgain);
            break;
        case "Warning":
            componentEvents.FireWarning(messageCode, subComponent, message, "", 0);
            break;
        case "Error":
            componentEvents.FireError(messageCode, subComponent, message, "", 0);
            break;
    }
}
```

Figure 18-1 The logMessage function

Add a process “starting” information message to the `Execute` method using the code in Listing 18-2:

```
string packagePath = "\\SSISDB\\" + PackageFolder + "\\"
                   + PackageProject + "\\"
                   + PackageName;
string msg = "Starting " + packagePath + " on " + ServerName;
logMessage(componentEvents, "Information", 1001, TaskName, msg);
```

Listing 18-2 Adding a process "starting" message

Once added, the code appears as shown in Figure 18-2:

```
public override DTSExecResult Execute(
    Connections connections,
    VariableDispenser variableDispenser,
    IDTSComponentEvents componentEvents,
    IDTSLogging log,
    object transaction)
{
    ConnectionManagerIndex = returnConnectionManagerIndex(connections, ConnectionManagerName);

    catalogProject = returnCatalogProject(ServerName, PackageFolder, PackageProject);
    catalogPackage = returnCatalogPackage(ServerName, PackageFolder, PackageProject, PackageName);

    Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet>
        executionValueParameterSet = returnExecutionValueParameterSet();

    string packagePath = "\\SSISDB\\" + PackageFolder + "\\"
          + PackageProject + "\\"
          + PackageName;
    string msg = "Starting " + packagePath + " on " + ServerName;
    logMessage(componentEvents, "Information", 1001, TaskName, msg);

    catalogPackage.Execute(Use32bit, null, executionValueParameterSet);

    return DTSExecResult.Success;
}
```

Figure 18-2 Adding a "starting" message

Add an "ended" message using the code in Listing 18-3:

```
msg = packagePath + " on " + ServerName + " completed";
logMessage(componentEvents, "Information", 1002, TaskName, msg);
```

Listing 18-3 Adding an "ended" message

Once added, the code appears as shown in Figure 18-3:

```
public override DTSExecResult Execute(
    Connections connections,
    VariableDispenser variableDispenser,
    IDTSComponentEvents componentEvents,
    IDTSLogging log,
    object transaction)
{
    ConnectionManagerIndex = returnConnectionManagerIndex(connections, ConnectionManagerName);

    catalogProject = returnCatalogProject(ServerName, PackageFolder, PackageProject);
    catalogPackage = returnCatalogPackage(ServerName, PackageFolder, PackageProject, PackageName);

    Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet>
        executionValueParameterSet = returnExecutionValueParameterSet();

    string packagePath = "\\SSISDB\\" + PackageFolder + "\\"
          + PackageProject + "\\"
          + PackageName;
    string msg = "Starting " + packagePath + " on " + ServerName;
    logMessage(componentEvents, "Information", 1001, TaskName, msg);

    catalogPackage.Execute(Use32bit, null, executionValueParameterSet);

    msg = packagePath + " on " + ServerName + " completed";
    logMessage(componentEvents, "Information", 1002, TaskName, msg);

    return DTSExecResult.Success;
}
```

Figure 18-3 Adding an "ended" message

Build and test the Execute Catalog Package Task, observing the Execution Results tab as shown in Figure 18-4:

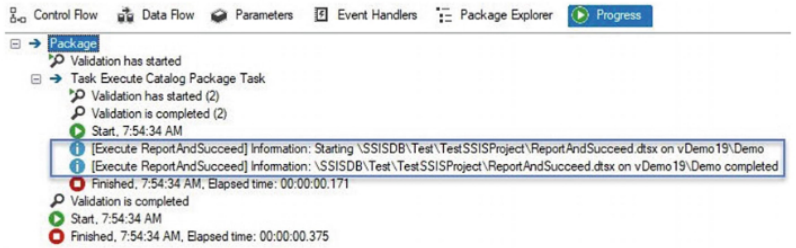


Figure 18-4 Observing the messages

The code executes and reports as designed, but the report code is designed inaccurately. When the Execute Catalog Package Task executes an SSIS package in the SSIS Catalog, the `Synchronized` execution parameter governs the behavior of the `package.Execute()` method. If `Synchronized` is false (which is the default), the SSIS Catalog finds the package and then returns from execution. If `Synchronized` is true, the SSIS Catalog executes the package *in-process*, returning from execution only *after* the SSIS package execution completes in the SSIS Catalog. The message should stand out, so change the `messageType` argument in the `logMessage` method from "Information" to "Warning."

The "ended" message is currently inaccurate. Update the "ended" message logic using the code in Listing 18-4:

```
msg = packagePath + " on " + ServerName
 + (Synchronized ? " completed." : " started. Check SSIS Catalog Reports
logMessage(componentEvents, "Warning", 1002, TaskName, msg);
```

Listing 18-4 Updating the "ended" message

Once added, the code appears as shown in Figure 18-5:

```
msg = packagePath + " on " + ServerName + (Synchronized ? " completed."
    : " started. Check SSIS Catalog Reports for package execution results.");
logMessage(componentEvents, "Warning", 1002, TaskName, msg);
```

Figure 18-5 Updated "ended" message and message type

Build the ExecuteCatalogPackageTask solution and add the Execute Catalog Package Task to a test SSIS project, configured to execute a package with the `Synchronized`

property set to False. When execution completes, view the Execution Results/Progress tab to find the newly updated (and more accurate) "ended" message, as shown in Figure 18-6:

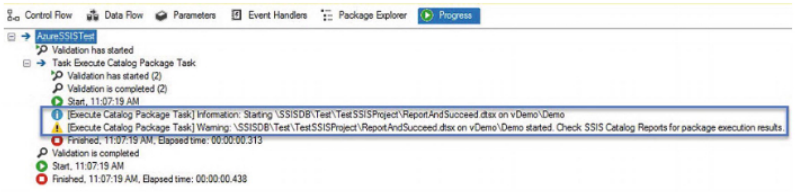


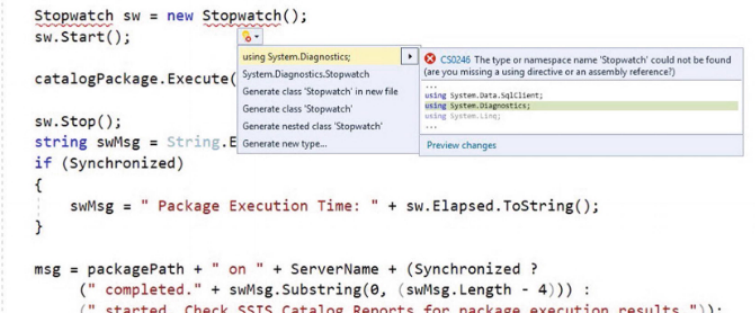
Figure 18-6 The new "ended" message

The "ended" message should also include execution time instrumentation, and the message could be flagged as a warning instead of information by implementing the code in Listing 18-5:

```
Stopwatch sw = new Stopwatch();
sw.Start();
.
.
.
sw.Stop();
string swMsg = String.Empty;
if(Synchronized)
{
  swMsg = " Package Execution Time: " + sw.Elapsed.ToString();
}
msg = packagePath + " on " + ServerName + (Synchronized ?
     (" completed." + swMsg.Substring(0, (swMsg.Length - 4))) :
     (" started. Check SSIS Catalog Reports for package execution result
logMessage(componentEvents, (Synchronized ? "Information" : "Warning"),
```

Listing 18-5 Adding execution time and warning

When added, the code appears as shown in Figure 18-7:



```
    logMessage(componentEvents, (Synchronized ? "Information" : "Warning")
        , 1002, TaskName, msg);

    return DTSExecResult.Success;
}
```

Figure 18-7 Updating the "ended" message, again

As before, a directive is missing but Visual Studio knows what we need, the `using System.Diagnostics;` directive. Use Quick Actions to add the `using System.Diagnostics;` directive. Once added, the code appears as shown in Figure 18-8:

```
92  Stopwatch sw = new Stopwatch();
93  sw.Start();
94
95  catalogPackage.Execute(Use32bit, null, executionValueParameterSet);
96
97  sw.Stop();
98  string swMsg = String.Empty;
99  if (Synchronized)
100 {
101     swMsg = " Package Execution Time: " + sw.Elapsed.ToString();
102 }
103
104 msg = packagePath + " on " + ServerName + (Synchronized ?
105      (" completed." + swMsg.Substring(0, (swMsg.Length - 4))) :
106      (" started. Check SSIS Catalog Reports for package execution results."));
107 logMessage(componentEvents, (Synchronized ? "Information" : "Warning")
108     , 1002, TaskName, msg);
109
110
111 return DTSExecResult.Success;
112 }
```

Figure 18-8 The updated message code, after adding the using System.Diagnostics; directive

`Stopwatch` is a member of the `System.Diagnostics` .Net Framework assembly that allows developers to measure the elapsed time between two events. The `Stopwatch` type variable named `sw` is declared and initialized on line 92. The `sw Stopwatch` variable is started on line 93, just before the `Execute` method for the `PackageInfo` variable named `catalogPackage` is called on line 95.

Immediately after the `Execute` method returns, the `Stopwatch` is stopped on line 97. A string variable named swMsg is declared and initialized to an empty string on line 98. If the package was executed with the `Synchronized` property set to true (line 99), the `swMsg string` variable is populated with a message to inform developers and operators of the package execution elapsed time captured by the `Stopwatch` variable named `sw`.

The `string` variable named `msg` is populated on lines 104–106 and begins with the value of the `packagePath` string variable concatenated with the literal string value `" on "`, followed by the value of the `ServerName` property.

If the `Synchronized` property set to true – meaning the code *waited* until the SSIS package execution was completed before reaching this line of code – `" completed."` and the value of the `swMsg string` variable are concatenated to the value of the `msg` variable. If the `Synchronized` property set to false – meaning the code found the SSIS package and continued (also known as "fire and forget") – `" started. Check SSIS Catalog Reports for package execution results."` is concatenated to the value of the `msg` variable.

The if-then-else functionality is controlled using the ternary conditional operator: `<expression> ? <if true> : <if false>`. The expression must evaluate to either true or false. The expression portion is simply the `Synchronized` property, which is a Boolean value. The "if true" operation of the ternary operation is `(" completed." + swMsg.Substring(0, (swMsg.Length - 4)))`, which builds a message informing developers and operators of the package execution elapsed time. The `swMsg.Length - 4` code truncates the elapsed time at milliseconds. The "if false" operation of the ternary operation is `(" started. Check SSIS Catalog Reports for package execution results."))`, which builds a message informing developers and operators that the package has started executing asynchronously and that they should look elsewhere for package execution results.

A different ternary conditional operation, found on line 107, checks the value of the `Synchronized` property to determine whether to return an Information or Warning messageType.

Let's Test It!

To test, build the task solution and configure an Execute Catalog package Task with the `Synchronized` property set to False. When execution completes, view the Execution Results/Progress tab to find the newly-updated-again "ended" message, as shown in Figure 18-9:

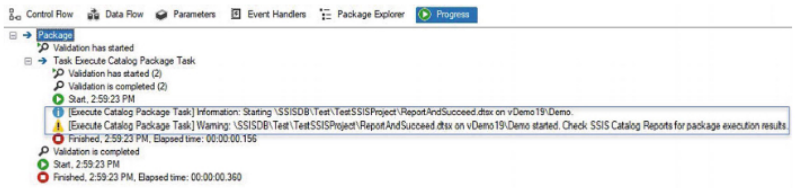


Figure 18-9 The new, new "ended" message

Execute Catalog Package Task instrumentation now surfaces "start" and "ended" messages.

Set the Execute Catalog Package Task Synchronized property to true, as shown in Figure 18-10:

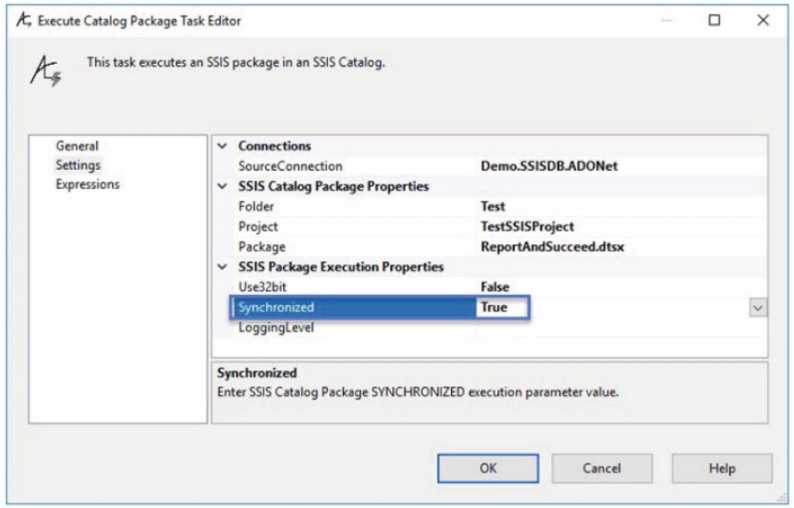


Figure 18-10 Setting the Execute Catalog Package Task Synchronized property to true

Execute the test SSIS package and view the Progress/Execution Results tab, shown in Figure 18-11:

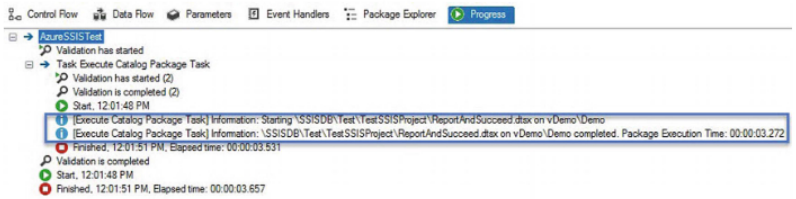


Figure 18-11 Synchronized execution instrumentation

Instrumentation is a great way to mitigate the total cost of ownership. Providing accurate and timely feedback to developers and operators reduces the time required to identify the root cause of a fault or failure.

Validation

Validation is especially important for preventing the attempted execution of improperly configured code. Validation is proactive and should respect property hierarchies. Let's start at the top of the SourceConnection ➤ Folder ➤ Project ➤ Package hierarchy by validating the SourceConnection property.

Validating the SourceConnection Property

We begin by testing for the existence of a value in the `SourceConnection` property and throwing an exception if no value exists. By way of review, the `SettingsNode.SourceConnection` property sets the `ExecuteCatalogPackageTask.ConnectionManagerName` property as shown in the code in Listing 18-6 (see also Listing 13-23):

```
[
  Category("Connections"),
  Description("The SSIS Catalog connection"),
  TypeConverter(typeof(ADONetConnections))
]
public string SourceConnection {
  get { return _task.ConnectionManagerName; }
  set { _task.ConnectionManagerName = value; }
}
```

Listing 18-6 The SettingsNode SourceConnection property

When the SourceConnection property changes, the code in `SettingsView.propertyGridSettings_PropertyValueChanged` updates the `ExecuteCatalogPackageTask.ConnectionManagerName` property using the code in Listing 18-7 (see also Listing 13-27):

```
theTask.ServerName = returnSelectedConnectionManagerDataSourceValue(sett
```

Listing 18-7 Updating the ExecuteCatalogPackageTask's ServerName property

The ExecuteCatalogPackageTask's `ConnectionManagerName` and `ServerName` properties are initialized as String.Empty as shown in Listing 18-8:

```
public string ConnectionManagerName { get; set; } = String.Empty;
.
.
.
public string ServerName { get; set; } = String.Empty;
```

Listing 18-8 SourceConnection-related properties initialized as String.Empty

The ExecuteCatalogPackageTask `Validate` method is called by the SSIS execution engine. Our validation code, therefore, belongs in the ExecuteCatalogPackageTask `Validate` method.

Begin validating the SourceConnection property by testing the `ConnectionManagerName` and `ServerName` property values for non-empty string values

using the code in Listing 18-9:

```
// test for SourceConnection (ConnectionManagerName and ServerName) exis
if((ConnectionManagerName == "") || (ServerName == ""))
{
  throw new Exception("Source Connection property is not configured.");
}
```

Listing 18-9 Begin validating SourceConnection

Once added, the `Validate()` appears as shown in Figure 18-12:

```
public override DTSExecResult Validate(
    Connections connections,
    VariableDispenser variableDispenser,
    IDTSComponentEvents componentEvents,
    IDTSLogging log)
{
    // test for SourceConnection (ConnectionManagerName and ServerName) existence
    if((ConnectionManagerName == "") || (ServerName == ""))
    {
        throw new Exception("Source Connection property is not configured.");
    }

    return DTSExecResult.Success;
}
```

Figure 18-12 The Validate method, updated

Build the Execute Catalog Package Task solution and test the updated version of the task by adding the Execute Catalog Package Task to a test SSIS package, as shown in Figure 18-13:

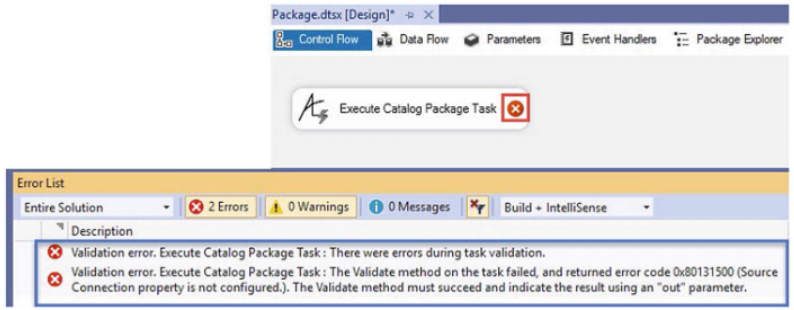


Figure 18-13 Testing the latest version of the Execute Catalog Package Task

Two changes are noteworthy:

1.The Execute Catalog Package Task now indicates an error state.

2.The Error List indicates two errors. The “root” error is listed second in the Error List window and reads: Validation error. Execute Catalog Package Task: The Validate method on the task failed, and returned error code 0x80131500 (Source Connection property is not configured.). The Validate method must succeed and indicate the result using an “out” parameter.

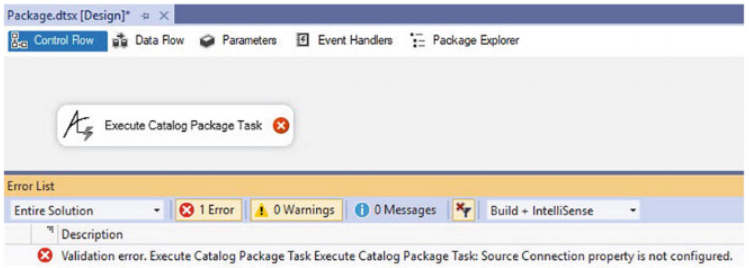


Figure 18-14 Cleaning up validation errors

Error-checking – and validation – code should *never* fail. Try-catch functionality elegantly manages failures in code execution. Rather than throwing exceptions in logic, we should leverage our recently added instrumentation functionality to inform developers of validation issues.

Refactor the Validate method using the code in Listing **18-10**:

```
public override DTSExecResult Validate(
           Connections connections,
           VariableDispenser variableDispenser,
           IDTSComponentEvents componentEvents,
           IDTSLogging log)
{
  // init validateResult
  DTSExecResult validateResult = DTSExecResult.Success;
  // test for SourceConnection (ConnectionManagerName and ServerName) ex
  try
  {
    if ((ConnectionManagerName == "") || (ServerName == ""))
    {
      logMessage(componentEvents
         , "Error"
         , -1001
         , TaskName
         , "Source Connection property is not configured.");
      validateResult = DTSExecResult.Failure;
    }
  }
```

```
  catch(Exception ex)
  {
    if(componentEvents != null)
    {
      // fire a generic error containing exception details
      logMessage(componentEvents
         , "Error"
         , -1001
         , TaskName
         , ex.Message);
    }
  // manage validateResult state
  validateResult = DTSExecResult.Failure;
  }
  return validateResult;
}
```

Listing 18-10 Refactoring Validate

Once refactored, the `Validate()` method appears as shown in Figure 18-15:

```
59  public override DTSExecResult Validate(
60      Connections connections,
61      VariableDispenser variableDispenser,
62      IDTSComponentEvents componentEvents,
63      IDTSLogging log)
64  {
65      // init validateResult
66      DTSExecResult validateResult = DTSExecResult.Success;
67
68      // test for SourceConnection (ConnectionManagerName and ServerName) existence
69      try
70      {
71          if ((ConnectionManagerName == "") || (ServerName == ""))
72          {
73              logMessage(componentEvents
74                  , "Error"
75                  , -1001
76                  , TaskName
77                  , "Source Connection property is not configured.");
78
79              validateResult = DTSExecResult.Failure;
80          }
81      }
82      catch (Exception ex)
83      {
84          if (componentEvents != null)
85          {
86              // fire a generic error containing exception details
87              logMessage(componentEvents
88                  , "Error"
89                  , -1001
90                  , TaskName
91                  , ex.Message);
92          }
93
94          // manage validation state
95          validateResult = DTSExecResult.Failure;
```

```
96      }
97
98      return validateResult;
99  }
```

Figure 18-15 Validate, refactored

The refactored `Validate()` method declares and initializes a `DTSExecResult` type variable named `validateResult` on line 66. The `validateResult` variable manages validation state throughout the `Validate()` method – see lines 79 and 95 – and is returned from the `Validate()` method on line 98.

A `try` block is initiated on line 69 and covers the code on lines 71–81.

The code on line 71 tests the `ConnectionManagerName` and `ServerName` properties for non-empty string values. If non-empty string values are detected in the `ConnectionManagerName` and `ServerName` properties, the code calls the `logMessage` function which fires an error event in the SSIS package.

The `catch` block starts on line 82 and covers the code on lines 83–96.

The code on line 84 checks to make sure the `componentEvents` argument (passed to the `Validate()` method) is not null. If not null, the `logMessage` function is called to raise an error event that contains details of any *other* error raised by the code in the `try` block in the SSIS package.

Taken together, this validation design pattern checks for a specific condition, answering the question, "Is the SourceConnection property configured with values?" If the SourceConnection property is *not* configured with values, an error is raised in the SSIS package. If any other error occurs when checking to see if the SourceConnection property is configured with values, the `catch` block raises an error event configured to report the error message. If the SourceConnection property *is* configured with values and no additional error occurs when checking to see if the SourceConnection property is configured with values, the `Validate()` method returns a `Success DTSExecResult` value.

As before, build the solution and test the updated ExecuteCatalogPackageTask functionality using a test SSIS package.

Once the SourceConnection property is configured, the `Validate()` method indicates no error, as shown in Figure **18-16**:

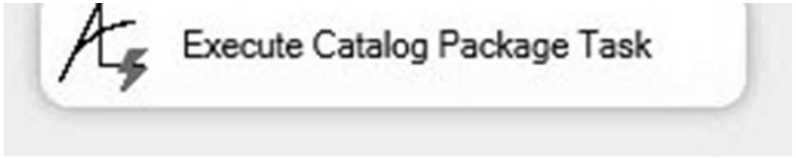


Figure 18-16 A validated SourceConnection property

The validation logic for the `ConnectionManagerName` and `ServerName` property values work as designed.

Building the Connection Validation Helper Function

The next step is to make sure the Execute Catalog Package Task can connect to the SSIS Catalog configured by the `ConnectionManagerName` and `ServerName` property values. Validating the connection will require several *helper* functions. Add the `returnConnection` and `attemptConnection` helper functions in Listing 18-11:

```
private SqlConnection returnConnection()
{
  return (SqlConnection)Connections[ConnectionManagerId].AcquireConnecti
}
private bool attemptConnection(Connections connections
                               , int connectionManagerIndex)
{
  bool ret = false;
  try
  {
    using (System.Data.SqlClient.SqlConnection con = returnConnection())
    {
      if (con.State != System.Data.ConnectionState.Open)
      {
        con.Open();
      }
      ret = true;
    }
  }
  catch (Exception ex)
  {
    ret = false;
  }
  return ret;
}
```

Listing 18-11 Connection validation helper functions: returnConnection and attemptCor

To support connection manager connection string overrides, returnConnection

To support connection manager connection string overrides, returnConnection returns a SqlConnection obtained by calling the AcquireConnection method for the Connections collection object defined using the ConnectionManagerId value. If the Connection Manager connects to an instance Azure SQL Database – which can require a username and password (SQL Login) – converting the AcquireConnection method return value to a SqlConnection may be the only way to acquire this connection.

The returnConnection method is used to set the connection used by the Execute Catalog Package Task when the attemptConnection method is called. The attempt-Connection is called early in the task properties Validate process.

Once added, the `returnConnection` and `attemptConnection` helper functions appear as shown in Figure 18-17:

```
149  private SqlConnection returnConnection()
150  {
151      return (SqlConnection)Connections[ConnectionManagerId].AcquireConnection(null);
152  }
153
     0 references | Andy Leonard, 7 days ago | 1 author, 8 changes
154  private bool attemptConnection(Connections connections
155                                 , int connectionManagerIndex)
156  {
157      bool ret = false;
158
159      try
160      {
161          using (System.Data.SqlClient.SqlConnection con = returnConnection())
162          {
163              if (con.State != System.Data.ConnectionState.Open)
164              {
165                  con.Open();
166              }
167              ret = true;
168          }
169      }
170      catch (Exception ex)
171      {
172          ret = false;
173      }
174
175      return ret;
176  }
```

Figure 18-17 The attemptConnection helper function

The return value for the `returnConnection` method is a `SqlConnection`. The code in the `returnConnection` method is a single line, but that line of code is *busy*. Let's unpack that line of code.

An SSIS Connection Manager's `AcquireConnection` method returns a `SqlConnection` type value. It is possible to connect to an SSIS Catalog hosted in Azure SQL DB. Later in the book, we will cover more about using the Execute Catalog Package Task to execute an SSIS package deployed to an Azure Data

Factory (ADF) SSIS Integration Runtime (IR), often referred to as "Azure-SSIS." Although SSIS Catalogs hosted on on-premises require Windows Authentication for interaction (deployment, execution), it's possible to interact with an SSIS Catalog hosted on Azure SQL DB using a SQL Login. Connection managers *never* surface passwords. One way – perhaps the only way – to establish a connection to Azure SQL DB using a login is to call the Connection Manager's `AcquireConnection` method.

The code on line 151 returns the connection manager's `AcquireConnection` method, to which it passes a `null` value for the `txn` (transaction) `object` argument. The connection manager is identified by the ConnectionManagerId – a unique identifier populated earlier in the response to changes in the `SourceConnection` property value detected in the `SettingsView propertyGridSettings_PropertyValueChanged` method. There are two calls to set the `ExecuteCatalogPackageTask ConnectionManagerId` property – one for when a new connection is created and one for when an existing connection manager is selected. The line of code that sets the `ConnectionManagerId` is the same for both use cases: `theTask.ConnectionManagerId = theTask.GetConnectionID(theTask.Connections, theTask.ConnectionManagerName);`. The `GetConnectionID` method is built into the `Microsoft.SqlServer.Dts.Runtime.Task` .Net Framework assembly.

The `returnConnection` method returns a `SqlConnection` type value, regardless of authentication method.

The return value for the `attemptConnection` method is a Boolean (`bool`) type variable named `ret`. The `ret` variable is declared and initialized to `false` on line 157. `ret` is set to `true` on line 167 if the code is able to open the connection on line 165 – or finds the connection already open on line 163. A try-catch block catches exceptions and sets `ret` to `false` on line 172 if an error occurs. `ret` is returned from the function on line 175.

Calling attemptConnection in the Validate Method

The next step is to add logic to the Validate method to test `SourceConnection` property connectivity to the SSIS Catalog by calling the `attemptConnection` helper function using the code in Listing **18-12**:

```
// attempt to connect
bool connectionAttempt = attemptConnection(connections
        , ConnectionManagerIndex);
```

```
if (!connectionAttempt)
{
  logMessage(componentEvents
           , "Error"
           , -1002
           , TaskName
           , "SQL Server Instance Connection attempt failed.");
  validateResult = DTSExecResult.Failure;
}
```

Listing 18-12 Calling attemptConnection

Once added, the code appears as shown in Figure **18-18**:

```
public override DTSExecResult Validate(
    Connections connections,
    VariableDispenser variableDispenser,
    IDTSComponentEvents componentEvents,
    IDTSLogging log)
{
    // init validateResult
    DTSExecResult validateResult = DTSExecResult.Success;

    // test for SourceConnection (ConnectionManagerName and ServerName) existence
    try
    {
        if ((ConnectionManagerName == "") || (ServerName == ""))
        {
            logMessage(componentEvents
               , "Error"
               , -1001
               , TaskName
               , "Source Connection property is not configured.");
            validateResult = DTSExecResult.Failure;

            // attempt to connect
            bool connectionAttempt = attemptConnection(connections
                , ConnectionManagerIndex);
            if (!connectionAttempt)
            {
                logMessage(componentEvents
                    , "Error"
                    , -1002
                    , TaskName
                    , "SQL Server Instance Connection attempt failed.");

                validateResult = DTSExecResult.Failure;
            }
        }
    }
    catch (Exception ex)
    {
        if (componentEvents != null)
        {
            // fire a generic error containing exception details
            logMessage(componentEvents
               , "Error"
               , -1001
               , TaskName
               , ex.Message);
```

```
        }

        // manage validateResult state
        validateResult = DTSExecResult.Failure;
    }

    return validateResult;
}
```

Figure 18-18 Calling the attemptConnection helper function

How I Tested the attemptConnection Validation

To test the attemptConnection helper function, build the ExecuteCatalogPackageTask solution. I provisioned a new SQL Server instance named "HadACatalog" on a virtual machine named vDemo19. I created an SSIS Catalog on the HadACatalog instance, as shown in Figure 18-19:

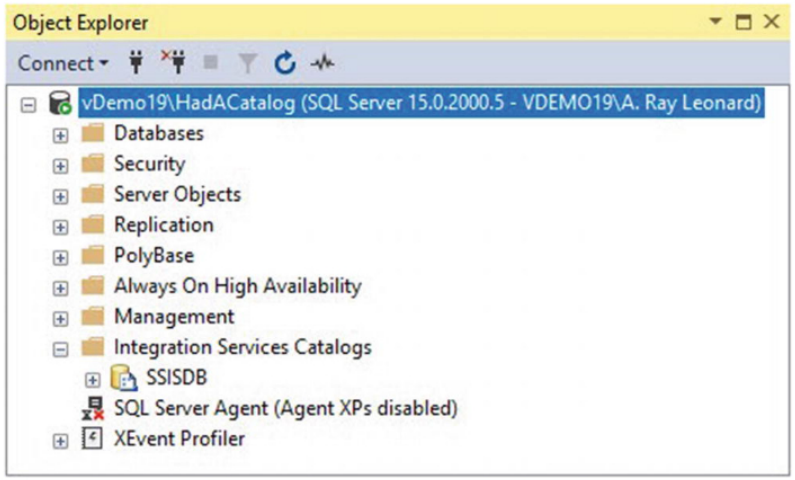


Figure 18-19 An SSIS Catalog on vDemo19\HadACatalog

I next configured an instance of the Execute Catalog Package Task in a test SSIS package to connect to the SSIS Catalog on the HadACatalog SQL Server instance, saved the test SSIS