A. Leonard, *Building Custom Tasks for SQL Server Integration Services*
https://doi.org/10.1007/978-1-4842-6482-9_15

15. Implement Use32bit, Synchronized, and LoggingLevel SettingsView Properties

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

In Chapters 13 and 14, we implemented `GeneralView` and `SettingsView`. In this chapter, we implement additional `SettingsNode` properties – Use32bit, Synchronized, and LoggingLevel. Adding the SourceConnection property was intense, so we begin this chapter by adding one straightforward and simple property implementation: User32bit. Be forewarned, the intensity increases.

Specifying a 32-Bit Interface

SSIS is provider-agnostic, which means SSIS can connect with just about any data provider installed on a server. If SSIS can connect to the provider, SSIS can read and write data through the said provider to resources for which the provider provides an interface.

Adding the Use32bit Property

Some providers surface 32-bit interfaces. A handful of providers surface *only* 32-bit interfaces. For this reason, the SSIS Catalog allows developers and operators to execute SSIS packages in 32-bit mode by setting a "use 32-bit" Boolean value to true at execution time.

To add the Use32bit property to `SettingsNode`, we begin by adding the code in Listing 15-1 to the `SettingsNode` class :

```
[
 Category("SSIS Package Execution Properties"),
 Description("Enter SSIS Catalog Package Use32bit execution property val
]
public bool Use32bit {
  get { return _task.Use32bit; }
```

```
    set { _task.Use32bit = value; }
  }
```

Listing 15-1 Adding the Use32bit property to SettingsNode

Once added, the `SettingsNode Use32bit` property code appears as shown in Figure 15-1:

```
[
    Category("SSIS Package Execution Properties"),
    Description("Enter SSIS Catalog Package Use32bit execution property value.")
]
0 references
public bool Use32bit {
    get { return _task.Use32bit; }
    set { _task.Use32bit = value; }
}
```

Figure 15-1 The SettingsNode Use32bit property, implemented

The next step is to update the `ExecuteCatalogPackageTask Execute` method to use the value set in the Use32bit property. Edit the `catalogPackage.Execute(false, null);` statement to match the code in Listing 15-2:

```
catalogPackage.Execute(Use32bit, null);
```

Listing 15-2 Configure the ExecuteCatalogPackageTask Execute method to use the Us

Once edited, the `ExecuteCatalogPackageTask Execute` method appears as shown in Figure 15-2:

```
public override DTSExecResult Execute(
    Connections connections,
    VariableDispenser variableDispenser,
    IDTSComponentEvents componentEvents,
    IDTSLogging log,
    object transaction)
{
    catalogServer = new Server(ServerName);
    integrationServices = new IntegrationServices(catalogServer);
    catalog = integrationServices.Catalogs[PackageCatalogName];
    catalogFolder = catalog.Folders[PackageFolder];
    catalogProject = catalogFolder.Projects[PackageProject];
    catalogPackage = catalogProject.Packages[PackageName];

    catalogPackage.Execute(Use32bit, null);

    return DTSExecResult.Success;
}
```

Figure 15-2 The ExecuteCatalogPackageTask Execute method now uses the Use32bit property

That's it. The Use32bit property is implemented. That wasn't so difficult, was it?

Test Use32bit

To test the Use32bit property, build the ExecuteCatalogPackageTask solution, open a test SSIS package, and configure the Execute Catalog Package Task to execute an SSIS package deployed to an SSIS Catalog. Set the Use32bit property to True, as shown in Figure 15-3:

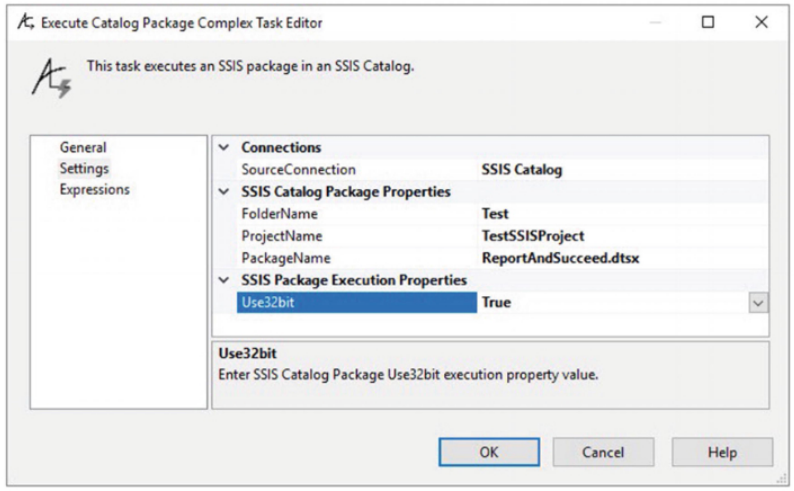

Figure 15-3 Setting the Use32bit property to True

Click the OK button and then execute the test SSIS package. If all goes as planned, the package execution will succeed in the SSIS Catalog. One way to check if the Use32bit property was set is to execute the T-SQL query in Listing 15-3:

```
Select folder_name
     , project_name
     , package_name
     , use32bitruntime
From SSISDB.catalog.executions
Order by execution_id DESC
```

Listing 15-3 Checking to see if Use32bit was set

Executing the T-SQL in Listing 15-3 returns results that confirm the Use32bit property works, as shown in Figure 15-4:

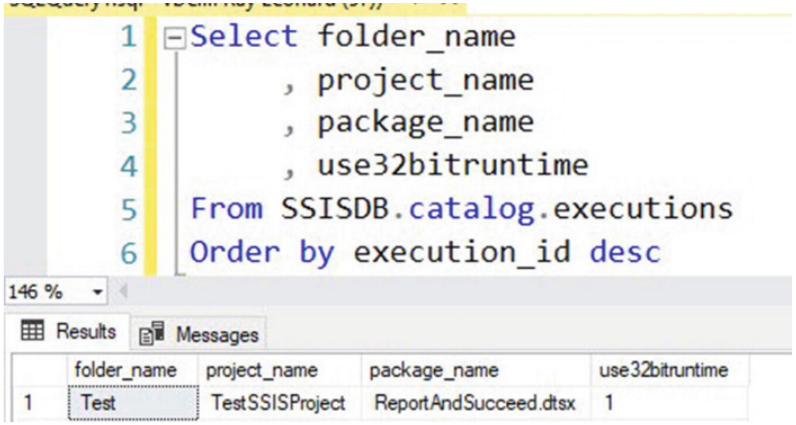

Figure 15-4 The Use32bit property in action

The next property is Synchronized, and the implementation is as easy as the implementation of the Use32bit property.

Adding the Synchronized Property

By default, SSIS packages executed in the SSIS Catalog execute with the Synchronized execution parameter set to *false*. A false Synchronized execution parameter means the SSIS Catalog execution engine finds and starts the SSIS package and then returns. It's possible for the SSIS package to continue executing – for hours or days – and then fail. People who monitor SSIS executions must check the status of the SSIS package execution to determine the state of the instance of execution. One good way to monitor the state of an instance of execution for an SSIS package is to view the SSIS Catalog All Executions and Overview reports built into SQL Server Management Studio (SSMS).

If one desires an SSIS package execution process to remain "in process," or to continue to show the process as running until the instance of execution *completes*, one may set the Synchronized execution parameter to true. To add the Synchronized property to `SettingsNode`, we begin by adding the code in Listing 15-4 to the `SettingsNode` class:

```
[
  Category("SSIS Package Execution Properties"),
  Description("Enter SSIS Catalog Package SYNCHRONIZED execution paramet
]
public bool Synchronized {
  get { return _task.Synchronized; }
  set { _task.Synchronized = value; }
}
```

Listing 15-4 Adding the Synchronized property to SettingsNode

Listing 15-4 Adding the Synchronized property to SettingsNode

Once added, the `SettingsNode Synchronized` property code appears as shown in Figure 15-5:

```
[
    Category("SSIS Package Execution Properties"),
    Description("Enter SSIS Catalog Package SYNCHRONIZED execution parameter value.")
]
0 references
public bool Synchronized {
    get { return _task.Synchronized; }
    set { _task.Synchronized = value; }
}
```

Figure 15-5 The SettingsNode Synchronized property, implemented

The next step is to update the `ExecuteCatalogPackageTask Execute` method to use the value set in the Synchronized property. Editing the `Execute` method is not as simple and straightforward as the edits to implement the Use32bit property; it's a little more involved. It all starts with a look at the `Execute` *statement* overloads in the `ExecuteCatalogPackageTask Execute` *method*.

Examining Execute() Overloads

To take a look at the available overloads for the SSIS package `Execute` method, find the `Execute` method in the `ExecuteCatalogPackageTask` code. Locate the line of code that calls the execution of the SSIS package – the line that we just updated when adding the Use32bit property. At this time, the line reads `catalogPackage.Execute(Use32bit, null);`. Select the opening parenthesis and type *another* opening parenthesis over it to open the overload options, as shown in Figure 15-6:

```
catalogPackage.Execute(Use32bit, null);
▲ 1 of 3 ▼ long Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.Execute(bool use32RuntimeOn64, EnvironmentReference reference)
```

Figure 15-6 Viewing Execute overload 1 of 3

By viewing the overloads list, we know the `Execute` function has three overloads because the overloads tooltip tells us, "1 of 3." We also know overload 1 of 3 takes two arguments:

- A `Boolean` value named `use32RuntimeOn64` – for which we supplied our `Boolean` property named `Use32bit`
- An `EnvironmentReference` value named `reference`

If we click the "down arrow" to the right of the 1 of 3, we can view overload 2 of 3, as shown in Figure 15-7:

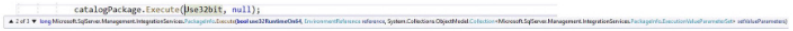

Figure 15-7 Viewing Execute overload 2 of 3

Viewing the overloads argument list for overload 2 of 3, we know this overload for the `Execute` takes three arguments, and the first two arguments are the same as overload 1 of 3:

- A `Boolean` value named `use32RuntimeOn64` – for which we supplied our `Boolean` property named `Use32bit`
- An `EnvironmentReference` value named `reference`
- A `Collection` of one or more `Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet` value(s) named `setValueParameters`

This third argument – the `collection` of `ExecutionValueParameterSet` parameters named `setValueParameters` – is the argument that will hold the `Synchronized` property value we are now coding. But first, let's look at the Execute 3 of overload.

If we click the "down arrow" to the right of the 2 of 3, we can view Execute overload 3 of 3, as shown in Figure 15-8:

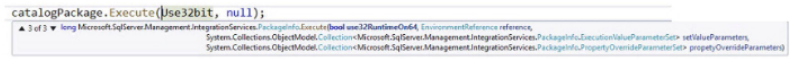

Figure 15-8 Viewing Execute overload 3 of 3

Viewing the overloads argument list for overload 3 of 3, we know this overload for the `Execute` takes four arguments, and the first three arguments are the same as overload 2 of 3:

- A `Boolean` value named `use32RuntimeOn64` – for which we supplied our `Boolean` property named `Use32bit`
- An `EnvironmentReference` value named `reference`
- A `Collection` of one or more `Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet` value(s) named `setValueParameters`
- A `Collection` of one or more `Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.PropertyOverrideParameterSet` value(s) named `propertyOverrideParameters`

We will use the second overload – 2 of 3 – in this version of the

`ExecuteCatalogPackageTask`.

There are two steps to adding the ExecutionValueParameterSet:

1. Build the ExecutionValueParameterSet.
2. Add the ExecutionValueParameterSet to the call to `package.Execute()`.

Building the ExecutionValueParameterSet

The `ExecutionValueParameterSet` is a collection of the `Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValuearameterSet` type. The first step is to create a function that builds and returns the `ExecutionValueParameterSet`, using the code in Listing 15-5:

```
using System.Collections.ObjectModel;
.
.
.
private Collection<Microsoft.SqlServer.Management.IntegrationServices.Pa
{
  // initialize the parameters collection
  Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageI
  // set SYNCHRONIZED execution parameter
  executionValueParameterSet.Add(new Microsoft.SqlServer.Management.Inte
  return executionValueParameterSet;
}
```

Listing 15-5 Building the returnExecutionValueParameterSet function

Add the `using System.Collections.ObjectModel;` directive to the ExecuteCatalogPackageTask.cs file header, along with other `using` directives.

Once added, the `returnExecutionValueParameterSet` function appears as shown in Figure 15-9:

```
82  private Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet> returnExecutionValueParameterSet()
83  {
84
85      // initialize the parameters collection
86      Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet> executionValueParameterSet =
87          new Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet>();
88
89      // set SYNCHRONIZED execution parameter
90      executionValueParameterSet.Add(
91          new Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet {
92              ObjectType = 50, ParameterName = "SYNCHRONIZED", ParameterValue = Synchronized });
93
94      return executionValueParameterSet;
95  }
```

Figure 15-9 The returnExecutionValueParameterSet function

The returnExecutionValueParameterSet function is declared to return a value of

type `Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet>` on line 82.

A new variable named `executionValueParameterSet` of `Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet>` type is created on lines 86–87.

On lines 90–92, a new `Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet>` object is initialized and added to the `executionValueParameterSet` collection.

On line 94, the `executionValueParameterSet` collection is returned to the caller, to which we now turn our attention.

Calling the returnExecutionValueParameterSet Function

Return to the `ExecuteCatalogPackageTask Execute` method, and overwrite the catalogPackage.Execute(Use32bit, null); statement with the code in Listing 15-6:

```
Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageInf
catalogPackage.Execute(Use32bit, null, executionValueParameterSet);
```

Listing 15-6 Updating package.Execute()

Once edited, the code near the end of the `ExecuteCatalogPackageTask Execute` method appears as shown in Figure 15-10:

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

catalogPackage.Execute(Use32bit, null, executionValueParameterSet);

return DTSExecResult.Success;
}
```

Figure 15-10 Package.Execute(), updated to configure and use execution parameters

The next step is to add the `Synchronized` Property to `SettingsNode`.

Adding the Synchronized Property to SettingsView

To add the Synchronized property to `SettingsNode`, begin by adding the code in Listing 15-7 to the `SettingsNode` class :

```
[
```

```
    Category("SSIS Package Execution Properties"),
    Description("Enter SSIS Catalog Package SYNCHRONIZED execution paramet
  ]
  public bool Synchronized {
    get { return _task.Synchronized; }
    set { _task.Synchronized = value; }
  }
```

Listing 15-7 Adding the Synchronized property to SettingsNode

Once added, the `SettingsNode Synchronized` property code appears as shown in Figure 15-11:

```
[
    Category("SSIS Package Execution Properties"),
    Description("Enter SSIS Catalog Package SYNCHRONIZED execution parameter value.")
]
0 references
public bool Synchronized {
    get { return _task.Synchronized; }
    set { _task.Synchronized = value; }
}
```

Figure 15-11 The SettingsNode Synchronized property, implemented

The next step is to test the `SettingsNode Synchronized` property.

Test Synchronized

To test the Synchronized property, build the ExecuteCatalogPackageTask solution, open a test SSIS package, and configure the Execute Catalog Package Task to execute an SSIS package deployed to an SSIS Catalog. Set the Synchronized property to True, as shown in Figure 15-12:

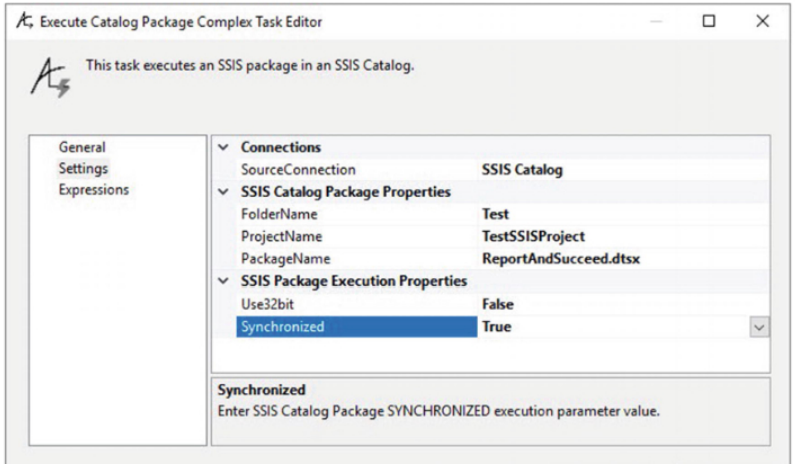

Figure 15-12 Setting the Synchronized property to True

Click the OK button and execute the test SSIS package in debug. If all goes as planned, the test SSIS package debug execution succeeds as shown in Figure **15-13**:

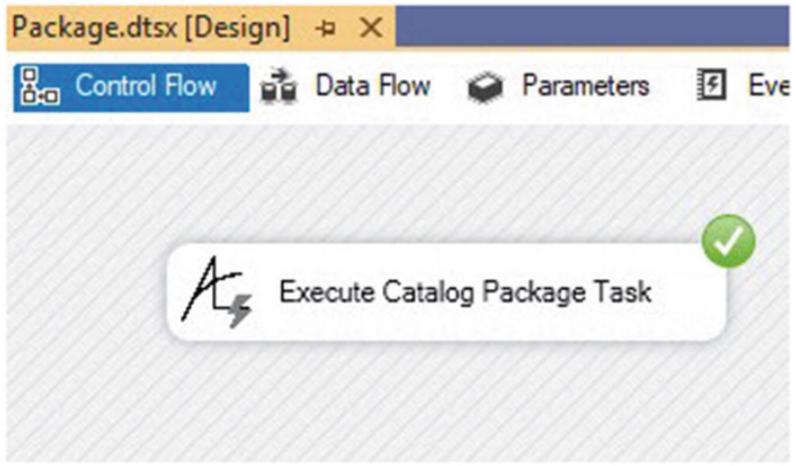

Figure 15-13 Successful test execution

Browse to the Overview report for this test execution in SSMS's Catalog Reports. The SYNCHRONIZED execution parameter – in the Parameters Used sub-report – should indicate the Synchronized execution parameter set to True, as shown in Figure **15-14**:

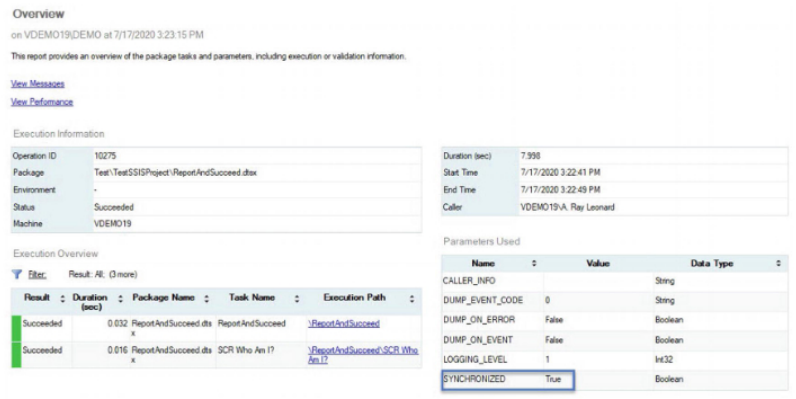

Figure 15-14 Synchronized execution parameter True

Did you notice each implementation grow in complexity? The next property we will implement is Logging Level

Adding the LoggingLevel Property

By default, SSIS packages executed in the SSIS Catalog execute with the Logging Level execution parameter set to *1* (Basic). Logging level is an enumeration of logging level name values that map to a list of integers. Do you smell something? I do. I smell another `TypeConverter` cooking…

Depending on the version of the SSIS Catalog in use, an SSIS developer may encounter four to five built-in logging level settings, and – perhaps – many more custom-defined logging levels. And logging level may be configured as the SSIS Catalog default logging level, but nearly all SSIS developers agree setting the default Catalog logging level to Basic is a best practice. Logging level may be altered for any SSIS package execution.

Building the LoggingLevel TypeConverter

Before adding the `LoggingLevel` property to `SettingsNode`, we need to add another `TypeConverter` class . TypeConverters follow a pattern that includes one type-specific function named `GetSpecializedObject` and three overridden functions: `GetStandardValues`, `GetStandardValuesExclusive`, and `GetStandardValuesSupported`. Begin by adding the code in Listing 15-8 to the `SettingsNode` class:

```
internal class LoggingLevels : StringConverter
{
  private object GetSpecializedObject(object contextInstance)
  {
    DTSLocalizableTypeDescriptor typeDescr = contextInstance as DTSLocal
    if (typeDescr == null)
    {
      return contextInstance;
    }
    return typeDescr.SelectedObject;
  }
  public override StandardValuesCollection GetStandardValues(ITypeDescri
  {
    object retrievalObject = GetSpecializedObject(context.Instance) as o
    return new StandardValuesCollection(getLoggingLevels(retrievalObject
  }
  public override bool GetStandardValuesExclusive(ITypeDescriptorContext
  {
    return true;
  }
```

```
    public override bool GetStandardValuesSupported(ITypeDescriptorContext
    {
      return true;
    }
  }
```

Listing 15-8 Adding the LoggingLevel TypeConverter (StringConverter)

Once added, the `SettingsNode LoggingLevel` StringConverter code appears as shown in Figure 15-15:

```
internal class LoggingLevels : StringConverter
{
    1 reference
    private object GetSpecializedObject(object contextInstance)
    {
        DTSLocalizableTypeDescriptor typeDescr = contextInstance as DTSLocalizableTypeDescriptor;

        if (typeDescr == null)
        {
            return contextInstance;
        }

        return typeDescr.SelectedObject;
    }

    1 reference
    public override StandardValuesCollection GetStandardValues(ITypeDescriptorContext context)
    {
        object retrievalObject = GetSpecializedObject(context.Instance) as object;

        return new StandardValuesCollection(getLoggingLevels(retrievalObject));
    }

    1 reference
    public override bool GetStandardValuesExclusive(ITypeDescriptorContext context)
    {
        return true;
    }

    1 reference
    public override bool GetStandardValuesSupported(ITypeDescriptorContext context)
    {
        return true;
    }
}
```

Figure 15-15 The LoggingLevels StringConverter class

All SSIS Catalog editions surface four logging levels:

- None (0)
- Basic (1)
- Performance (2)
- Verbose (3)

In all versions of the SSIS Catalog, Basic (1) is the default logging level. Complete the `LoggingLevels StringConverter` class by adding the `getLoggingLevels` function code in Listing 15-9:

```
private ArrayList getLoggingLevels(object retrievalObject)
{
  ArrayList logLevelsArray = new ArrayList();
  logLevelsArray.Add("None");
  logLevelsArray.Add("Basic");
  logLevelsArray.Add("Performance");
  logLevelsArray.Add("Verbose");
  return logLevelsArray;
}
```

Listing 15-9 Adding the getLoggingLevels function

Extending the ExecutionValueParameterSet with LoggingLevel

As we mentioned when adding the `Synchronized` execution parameter, the `ExecutionValueParameterSet` is a collection of the `Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValuearameterSet` type . Like the `Synchronized` execution parameter, `LoggingLevel` is also an execution parameter.

Before adding the LoggingLevel integer value to the execution parameters collection, the value must be decoded using the code in Listing **15-10**:

```
private int decodeLoggingLevel(string loggingLevel)
{
  int ret = 1;
  switch (loggingLevel)
  {
    default:
      break;
    case "None":
      ret = 0;
      break;
    case "Performance":
      ret = 2;
      break;
    case "Verbose":
      ret = 3;
      break;
  }
  return ret;
}
```

Listing 15-10 Adding the DecodeLoggingLevel function

When the `decodeLoggingLevel function` is added to the ExecuteCatalogPackageTask class, the code appears as shown in Figure 15-16:

```
private int decodeLoggingLevel(string loggingLevel)
{
    int ret = 1;

    switch (loggingLevel)
    {
        default:
            break;
        case "None":
            ret = 0;
            break;
        case "Performance":
            ret = 2;
            break;
        case "Verbose":
            ret = 3;
            break;
    }

    return ret;
}
```

Figure 15-16 The decodeLoggingLevel function

To add the LoggingLevel value to the returnExecutionValueParameterSet function, edit the returnExecutionValueParameterSet function to match the code in Listing 15-11:

```
private Collection<Microsoft.SqlServer.Management.IntegrationServices.Pa
{


  // initialize the parameters collection
  Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageI
  // set SYNCHRONIZED execution parameter
  executionValueParameterSet.Add(new Microsoft.SqlServer.Management.Inte
  // set LOGGING_LEVEL execution parameter
  int LoggingLevelValue = decodeLoggingLevel(LoggingLevel);
  executionValueParameterSet.Add(new Microsoft.SqlServer.Management.Inte
  return executionValueParameterSet;
}
```

Listing 15-11 Editing the returnExecutionValueParameterSet function

Once added, the `returnExecutionValueParameterSet` function appears as shown in Figure 15-17:

Figure 15-17 The returnExecutionValueParameterSet function, edited

The next step is to add the `SettingsView LoggingLevel` property.

Adding the LoggingLevel Property to SettingsView

To add the LoggingLevel property to `SettingsNode`, begin by adding the code in Listing 15-12 to the `SettingsNode` class :

```
[
  Category("SSIS Package Execution Properties"),
  Description("Enter SSIS Catalog Package LOGGING_LEVEL execution parame
  TypeConverter(typeof(LoggingLevels))
]
public string LoggingLevel {
  get { return _task.LoggingLevel; }
  set { _task.LoggingLevel = value; }
}
```

Listing 15-12 Adding the LoggingLevel property to SettingsNode

Once added, the `SettingsNode LoggingLevel` property code appears as shown in Figure 15-18:

Figure 15-18 The SettingsNode LoggingLevel property, implemented

The next step is to test the `SettingsNode LoggingLevel` property.

Test LoggingLevel

To test the LoggingLevel property, build the ExecuteCatalogPackageTask solution, open a test SSIS package, and configure the Execute Catalog Package Task to execute an SSIS package deployed to an SSIS Catalog. Set the LoggingLevel property to Verbose, as shown in Figure 15-19:

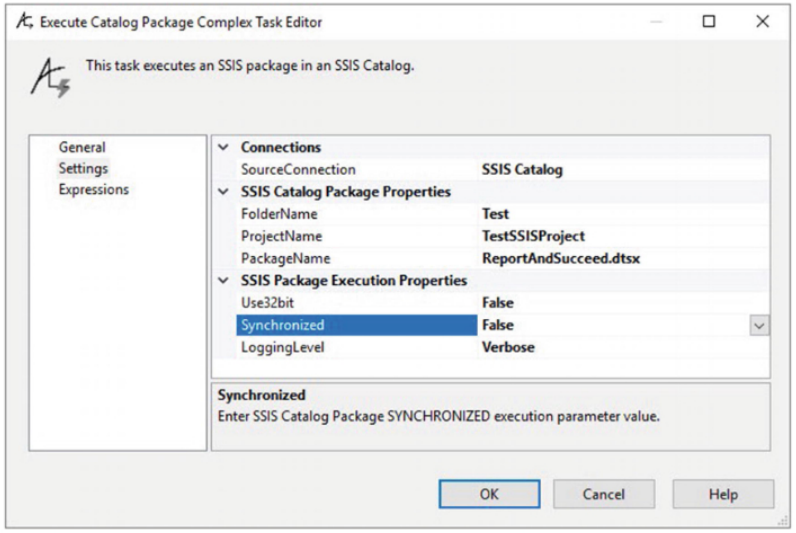

Figure 15-19 Setting the LoggingLevel property to Verbose

Click the OK button and execute the test SSIS package in debug. If all goes as planned, the test SSIS package debug execution succeeds as shown in Figure 15-20:

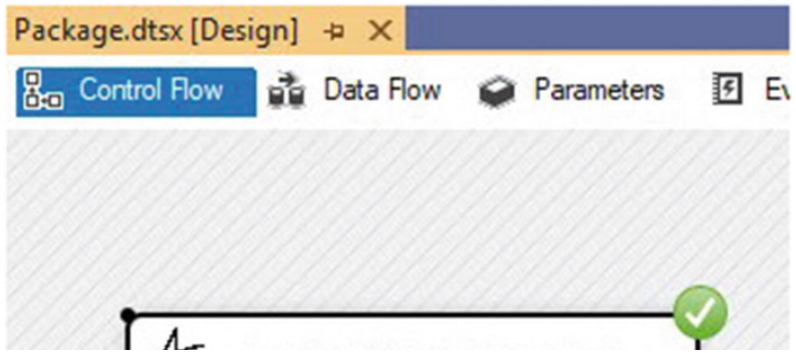

Figure 15-20 Successful test execution

Browse to the Overview report for this test execution in SSMS's Catalog Reports. The LOGGING_LEVEL execution parameter – in the Parameters Used sub-report – should indicate the LoggingLevel execution parameter set to 3 (Verbose), as shown in Figure 15-21:

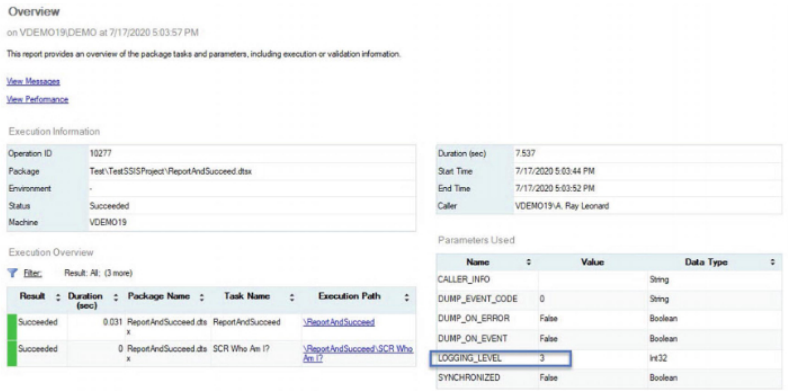

Figure 15-21 LoggingLevel execution parameter set to 3 (Verbose)

The next property to implement is Reference, which is the subject of the next chapter.

Conclusion

In this chapter, we implemented the `Use32bit`, `Synchronized`, and `LoggingLevel` `SettingsView` properties.

Now would be a good time to check in your code.

Previous chapter
‹ 14. Implement New Connection

Next chapter
16. Refactoring SourceConne ›