A. Leonard, *Building Custom Tasks for SQL Server Integration Services*
https://doi.org/10.1007/978-1-4842-6482-9_20

20. Adding Synchronous Execution Properties

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

Chapter 19 included solutions for two egregious bugs:

1. If the SSIS package executes for longer than 30 seconds, the `PackageInfo Execute()` returns an error message that reads: "The Execute method on the task returned error code 0x80131904 (Execution Timeout Expired. The timeout period elapsed prior to completion of the operation or the server is not responding.). The Execute method must succeed, and indicate the result using an 'out' parameter."
2. The Execute Catalog Package Task reports success, even when the SSIS package execution fails.

The solution for the first bug involves asynchronous execution. Three settings required for the `WaitHandle.WaitAny` solution are

- `maximumRetries`: An `int` variable that sets the number of times the timer is allowed to "tick" before the SSIS Catalog Package execution operation is terminated.
- `retryIntervalSeconds`: An `int` variable that sets the number seconds between timer "ticks," or retries.
- `operationTimeoutMinutes`: An `int` variable that sets the number of minutes permitted before the SSIS Catalog Package execution operation is terminated.

At this point in development, the `maximumRetries`, `retryIntervalSeconds`, and `operationTimeoutMinutes` variables and their values are hard-coded into the `executeSynchronous` helper function in the `ExecuteCatalogPackageTask` class. The `maximumRetries`, `retryIntervalSeconds`, and `operationTimeoutMinutes` variables should really be `ExecuteCatalogPackageTask` class properties so SSIS

developers may configure these values when Synchronous SSIS package execution exceeds 30 seconds.

In addition, the property values are related in that `maximumRetries` * `retryIntervalSeconds` should be less than the value `operationTimeoutMinutes` / 60 in order to avoid execution timeouts.

Surfacing the New Properties

The new properties should be visually linked to the `Synchronized` property setting in `SettingsNode`, and the `Synchronized` property should be separated from the "SSIS Package Execution Properties" category in `SettingsNode`. Begin by editing the `Synchronized` property in `SettingsNode` to appear in its own category named "SSIS Package Synchronized Properties" using the code in Listing 20-1:

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

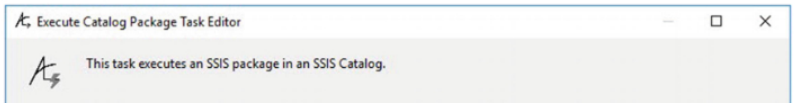
Listing 20-1 Editing the Synchronized property category

Once edited, the code appears as shown in Figure 20-1:

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

Figure 20-1 The Synchronized property is now in the SSIS Package Synchronized Properties category

Build the ExecuteCatalogPackageTask solution and then add the Execute Catalog Package Task to a test SSIS package. Open the Execute Catalog Package Task editor, and note the new location of the `Synchronized` property, as shown in Figure 20-2:



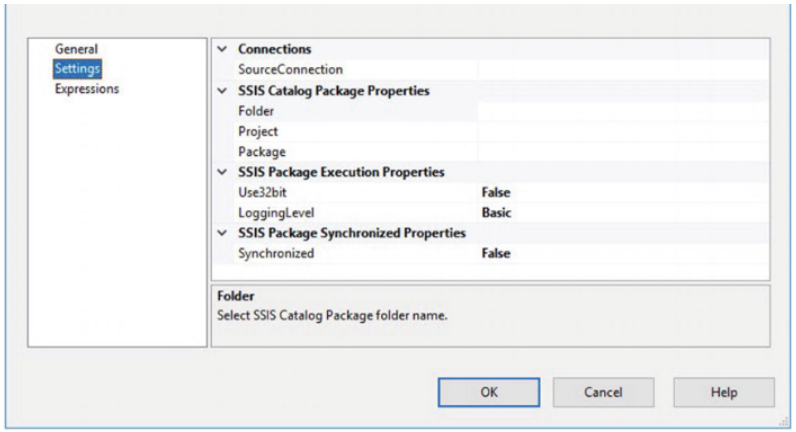


Figure 20-2 The Synchronized property, edited

The `Synchronized` property now has its own property category named "SSIS Package Synchronized Properties."

The next step is to add properties to manage the maximumRetries, retryIntervalSeconds, and operationTimeoutMinutes values.

Adding MaximumRetries, RetryIntervalSeconds, and OperationTimeoutMinutes

Add new properties to the ExecuteCatalogPackageTask object using the code in Listing 20-2:

```
public int MaximumRetries { get; set; } = 29;
public int RetryIntervalSeconds { get; set; } = 10;
public int OperationTimeoutMinutes { get; set; } = 5;
```

Listing 20-2 Adding MaximumRetries, RetryIntervalSeconds, and OperationTimeoutMin

When added, the code appears as shown in Figure 20-3:

```
public int MaximumRetries { get; set; } = 29;
public int RetryIntervalSeconds { get; set; } = 10;
public int OperationTimeoutMinutes { get; set; } = 5;
```

Figure 20-3 MaximumRetries, RetryIntervalSeconds, and OperationTimeoutMinutes properties, added to the ExecuteCatalogPackageTask class

Please note the default value for `MaximumRetries` (29) multiplied by the default value for `RetryIntervalSeconds` (10) equals 290. When applied to the SSIS package execution process, the timer will "tick" every 10 seconds 29 times before "timing out." The `OperationTimeoutMinutes` default is 5, or 300 seconds. The timer operation timeout is greater than `MaximumRetries` * `RetryIntervalSeconds`, which is a good place to start.

Next, surface the MaximumRetries, RetryIntervalSeconds, and OperationTimeoutMinutes properties on `SettingsNode` by adding the code in Listing 20-3 to the SettingsView.cs file:

```
[
  Category("SSIS Package Synchronized Properties"),
  Description("Enter SSIS Catalog Package Maximum Retries " +
              "for the timer \"tick\" when SYNCHRONIZED is true.")
]
public int MaximumRetries {
  get { return _task.MaximumRetries; }
  set { _task.MaximumRetries = value; }
}
[
  Category("SSIS Package Synchronized Properties"),
  Description("Enter SSIS Catalog Package Retry Interval Seconds " +
         "to wait between timer \"tick\" retries when SYNCHRONIZED is tr
]
public int RetryIntervalSeconds {
  get { return _task.RetryIntervalSeconds; }
  set { _task.RetryIntervalSeconds = value; }
}
[
  Category("SSIS Package Synchronized Properties"),
  Description("The SSIS Catalog Package Operation Timeout Minutes - " +
            "managed by RetryIntervalSeconds and MaximumRetires when SYN
            ReadOnly(true)
]
public int OperationTimeoutMinutes {
  get { return _task.OperationTimeoutMinutes; }
  set { _task.OperationTimeoutMinutes = value; }
}
```

Listing 20-3 Surfacing the MaximumRetries, RetryIntervalSeconds, and OperationTi

Once added, the code appears as shown in Figure 20-4:

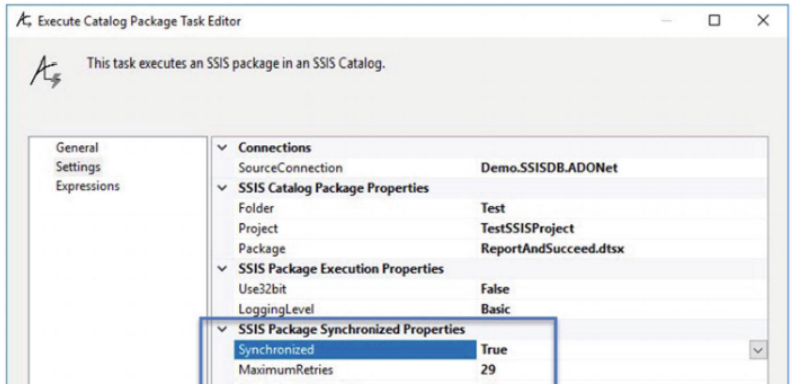
```
[
    Category("SSIS Package Synchronized Properties"),
```

```
    Description("Enter SSIS Catalog Package Maximum Retries " +
    "for the timer \"tick\" when SYNCHRONIZED is true.")
]
0 references
public int MaximumRetries {
    get { return _task.MaximumRetries; }
    set { _task.MaximumRetries = value; }
}

[
    Category("SSIS Package Synchronized Properties"),
    Description("Enter SSIS Catalog Package Retry Interval Seconds " +
    "to wait between timer \"tick\" retries when SYNCHRONIZED is true.")
]
0 references
public int RetryIntervalSeconds {
    get { return _task.RetryIntervalSeconds; }
    set { _task.RetryIntervalSeconds = value; }
}

[
    Category("SSIS Package Synchronized Properties"),
    Description("The SSIS Catalog Package Operation Timeout Minutes - " +
    "managed by RetryIntervalSeconds and MaximumRetires when SYNCHRONIZED is true."),
    ReadOnly(true)
]
1 reference
public int OperationTimeoutMinutes {
    get { return _task.OperationTimeoutMinutes; }
    set { _task.OperationTimeoutMinutes = value; }
}
```

Figure 20-4 Surfacing the MaximumRetries, RetryIntervalSeconds, and OperationTimeoutMinutes properties

Note the OperationTimeoutMinutes property ReadOnly attribute is true, which means the value appears in the Execute Catalog Package Task editor, but is not editable. The OperationTimeoutMinutes property is managed mathematically and available for viewing only.

Test the new properties by building the ExecuteCatalogPackageTask solution, opening a test SSIS package, adding an Execute Catalog Package Task, and configuring the new MaximumRetries, RetryIntervalSeconds, and OperationTimeoutMinutes properties on the Settings page, as shown in Figure 20-5:



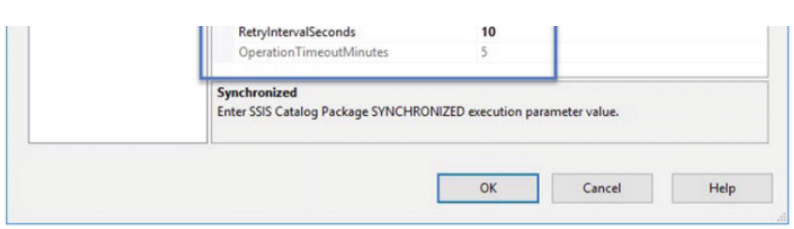


Figure 20-5 Testing the new MaximumRetries, RetryIntervalSeconds, and OperationTimeoutMinutes properties

Configure the Execute Catalog Package Task to execute a short-running (less than 5 minutes) SSIS package. Set the Synchronized property to True (because the new `MaximumRetries`, `RetryIntervalSeconds`, and `OperationTimeoutMinutes` properties are not used unless the Synchronized property is set to True), and then close the editor. Execute the test SSIS package. If all goes as planned, the test SSIS package should succeed when executed in the debugger, as shown in Figure 20-6:

Figure 20-6 Successful execution

The next step is to use the values of the `MaximumRetries` and `RetryIntervalSeconds` properties to manage the `OperationTimeoutMinutes` property. The `OperationTimeoutMinutes` property value is set by multiplying `maximumRetries` * `retryIntervalSeconds` to calculate projected execution seconds, and then the execution seconds value is stored in a variable named `retriesTimesIntervalSeconds`. The value of OperationTimeoutMinutes is calculated by dividing the value of the `retriesTimesIntervalSeconds` variable by 60, converting the result to an `int` value, and then adding 1 for good measure. Configure the `propertyGridSettings_PropertyValueChanged` method to respond to a change in either the `MaximumRetires` or the `RetryIntervalSeconds` property and trigger recalculation of the OperationTimeoutMinutes property value using the code in Listing 20-4:

```
if ((e.ChangedItem.PropertyDescriptor.Name.CompareTo("MaximumRetries") =
 || (e.ChangedItem.PropertyDescriptor.Name.CompareTo("RetryIntervalSecon
{
  int retriesTimesIntervalSeconds = settingsNode.MaximumRetries
       * settingsNode.RetryIntervalSeconds;
  int operationTimeoutSeconds = settingsNode.OperationTimeoutMinutes * 6
  settingsNode.OperationTimeoutMinutes = (int)((retriesTimesIntervalSeco
```

```
    this.settingsPropertyGrid.Refresh();
}
```

Listing 20-4 Responding to changes in the MaximumRetries and RetryIntervalSecond

When added to the `propertyGridSettings_PropertyValueChanged` method, the code appears as shown in Figure 20-7:

```
if ((e.ChangedItem.PropertyDescriptor.Name.CompareTo("MaximumRetries") == 0)
    || (e.ChangedItem.PropertyDescriptor.Name.CompareTo("RetryIntervalSeconds") == 0))
{
    int retriesTimesIntervalSeconds = settingsNode.MaximumRetries
        * settingsNode.RetryIntervalSeconds;
    int operationTimeoutSeconds = settingsNode.OperationTimeoutMinutes * 60;
    settingsNode.OperationTimeoutMinutes = (int)((retriesTimesIntervalSeconds / 60) + 1);
    this.settingsPropertyGrid.Refresh();
}
```

Figure 20-7 Auto-reset the OperationTimeoutMinutes property value

One could just as easily make the `MaximumRetries` property read-only and manage the `MaximumRetries` property value by calculating the `MaximumRetries` property value using the values of the `RetryIntervalSeconds` and the `OperationTimeoutMinutes` properties. The goal is the same: we want to surface wait-related properties in the Execute Catalog Package Task editor so SSIS developers who use the Execute Catalog Package Task can configure the wait-related properties to stop execution if the SSIS package has not completed execution in some amount of time.

Cleaning Up Outdated Asynchronous Execution

The next step is to clean up the (now) outdated local `maximumRetries`, `retryIntervalSeconds`, and `operationTimeoutMinutes` variables in the `executeSynchronous` method. Delete the code highlighted in Figure 20-8:

```
private Operation.ServerOperationStatus executeSynchronous(long executionId
                    , SqlConnection connection)
{
    Operation.ServerOperationStatus ret = Operation.ServerOperationStatus.UnexpectTerminated;
    ManualResetEvent manualResetState = new ManualResetEvent(false);

    int maximumRetries = 20;
    int retryIntervalSeconds = 10;
    int operationTimeoutMinutes = 10;
```

Figure 20-8 Deleting local variables

Once the local asynchronous-related variables are deleted, the `executeSynchronous` method appears as shown in Figure 20-9:

```
private Operation.ServerOperationStatus executeSynchronous(long executionId
```

```
                , SqlConnection connection)
{
    Operation.ServerOperationStatus ret = Operation.ServerOperationStatus.UnexpectTerminated;
    ManualResetEvent manualResetState = new ManualResetEvent(false);

    CheckWaitState statusChecker = new CheckWaitState(executionId
                                                , maximumRetries
                                                , connection
                                                , PackageCatalogName
                                                , this);
    TimeSpan dueTime = new TimeSpan(0, 0, 0);
    TimeSpan period = new TimeSpan(0, 0, retryIntervalSeconds);

    object timerState = new TimerCheckerState(manualResetState);
    TimerCallback timerCallback = statusChecker.CheckStatus;
    Timer timer = new Timer(timerCallback, timerState, dueTime, period);

    WaitHandle[] manualResetStateWaitHandleCollection = new WaitHandle[]
              { manualResetState };
    int timeoutMillseconds = (int)new TimeSpan(0, operationTimeoutMinutes
                                             , 0).TotalMilliseconds;
    bool exitContext = false;

    // wait here, please
    WaitHandle.WaitAny(manualResetStateWaitHandleCollection
                    , timeoutMillseconds
                    , exitContext);

    ret = statusChecker.OperationStatus;

    manualResetState.Dispose();
    timer.Dispose();

    return ret;
}
```

Figure 20-9 The executeSynchronous method after variables are deleted

In the `executeSynchronous` method, replace `maximumRetries` with `MaximumRetries`, `retryIntervalSeconds` with `RetryIntervalSeconds`, and `operationTimeoutMinutes` with `OperationTimeoutMinutes` using the code in Listing **20-5**:

```
private Operation.ServerOperationStatus executeSynchronous(long executio
                                                , SqlConnection connec
{
  Operation.ServerOperationStatus ret = Operation.ServerOperationStatus.
  ManualResetEvent manualResetState = new ManualResetEvent(false);
  CheckWaitState statusChecker = new CheckWaitState(executionId
                                                , MaximumRetries
                                                , connection
                                                , PackageCatalogName
                                                , this);
  TimeSpan dueTime = new TimeSpan(0, 0, 0);
  TimeSpan period = new TimeSpan(0, 0, RetryIntervalSeconds);
  object timerState = new TimerCheckerState(manualResetState);
  TimerCallback timerCallback = statusChecker.CheckStatus;
  Timer timer = new Timer(timerCallback, timerState, dueTime, period);
  WaitHandle[] manualResetStateWaitHandleCollection = new WaitHandle[]
                        { manualResetState };
```

```
    int timeoutMillseconds = (int)new TimeSpan(0, OperationTimeoutMinutes
                                                , 0).TotalMilliseconds;
    bool exitContext = false;
    // wait here, please
    WaitHandle.WaitAny(manualResetStateWaitHandleCollection
                              , timeoutMillseconds
                              , exitContext);
    ret = statusChecker.OperationStatus;
    manualResetState.Dispose();
    timer.Dispose();
    return ret;
  }
```

Listing 20-5 Updating local variables with new properties

Once updated, the code appears as shown in Figure **20-10**:

```
private Operation.ServerOperationStatus executeSynchronous(long executionId
                    , SqlConnection connection)
{
    Operation.ServerOperationStatus ret = Operation.ServerOperationStatus.UnexpectTerminated;
    ManualResetEvent manualResetState = new ManualResetEvent(false);

    CheckWaitState statusChecker = new CheckWaitState(executionId
                                            , MaximumRetries
                                            , connection
                                            , PackageCatalogName
                                            , this);
    TimeSpan dueTime = new TimeSpan(0, 0, 0);
    TimeSpan period = new TimeSpan(0, 0, RetryIntervalSeconds);

    object timerState = new TimerCheckerState(manualResetState);
    TimerCallback timerCallback = statusChecker.CheckStatus;
    Timer timer = new Timer(timerCallback, timerState, dueTime, period);

    WaitHandle[] manualResetStateWaitHandleCollection = new WaitHandle[]
              { manualResetState };
    int timeoutMillseconds = (int)new TimeSpan(0, OperationTimeoutMinutes
                                            , 0).TotalMilliseconds;
    bool exitContext = false;

    // wait here, please
    WaitHandle.WaitAny(manualResetStateWaitHandleCollection
                    , timeoutMillseconds
                    , exitContext);

    ret = statusChecker.OperationStatus;

    manualResetState.Dispose();
    timer.Dispose();

    return ret;
}
```

Figure 20-10 The updated executeSynchronous method

Let's Test It!

Build the ExecuteCatalogPackageTask solution, open a test SSIS project and package, and

then add a new Execute Catalog Package Task to the Control Flow. Configure the Execute Catalog Package Task to execute an SSIS package that takes some time to run, such as the RunForSomeTime.dtsx package, and set the Synchronized property to True, the MaximumRetries property to 50, and the RetryIntervalSeconds property to 2, as shown in Figure 20-11:

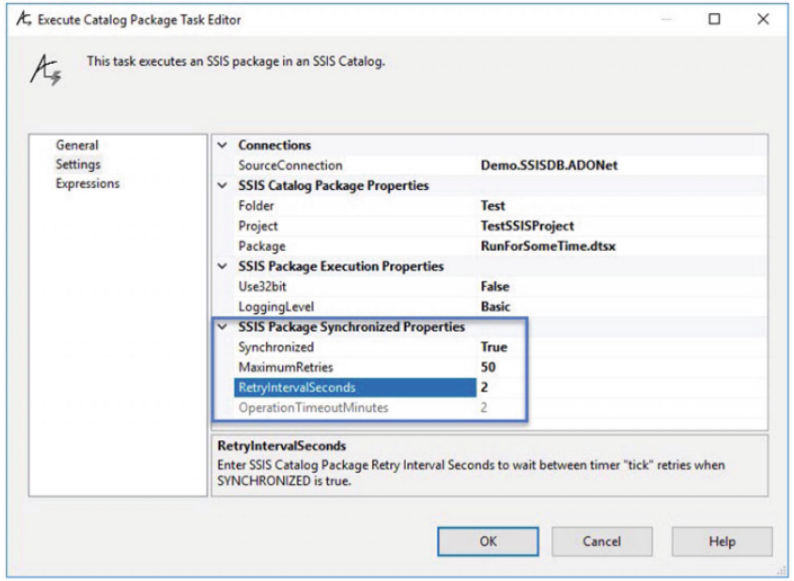


Figure 20-11 Configuring the Execute Catalog Package Task for a timer instrumentation test

Execute the test SSIS package and view the Progress tab. Timer "tick" instrumentation messages should appear similar to those shown in Figure 20-12:

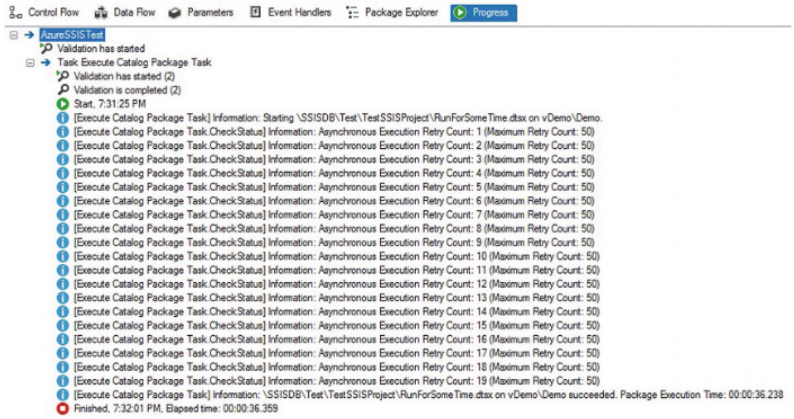


Figure 20-12 Timer "tick" instrumentation messages

Timer instrumentation is code complete.

Conclusion

In this chapter, we converted three hard-coded variables – `maximumRetries`, `retryIntervalSeconds`, and `operationTimeoutMinutes` – into `ExecuteCatalogPackageTask` class properties and then added code to the `SettingsView` and `SettingsNode` classes to surface the new properties to SSIS developers.

In the next chapter, we test the Execute Catalog Package Task functionality.

Now would be an excellent time to check in your code.

Previous chapter
< 19. Crushing Bugs

Next chapter
21. Testing the Task >