A. Leonard, *Building Custom Tasks for SQL Server Integration Services*
https://doi.org/10.1007/978-1-4842-6482-9_19

19. Crushing Bugs

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

Two egregious bugs remain in the Execute Catalog Package Task code:

1. The Execute Catalog Package Task reports success, even when the SSIS package execution fails.
2. If the SSIS package executes for longer than 30 seconds, the `ExecuteCatalogPackageTask.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."

The root cause of both bugs lies in the `Execute` method, so the solution is blended. Let's start with the second bug, Execution Timeout Expired.

Execution Timeout Expired

Why does the `ExecuteCatalogPackageTask.Execute` method time out at 30 seconds? I do not know. Just because I do not know or understand why some functionality is coded the way it is coded does *not* mean the functionality is coded incorrectly. I simply do not know why the `ExecuteCatalogPackageTask.Execute` method times out at 30 seconds. I *do*, however, need to work around this limitation.

The solution I found uses the `WaitHandle.WaitAny` method.

Threading, Briefly

The `WaitHandle.WaitAny` method is a threading topic, which is part of the `System.Threading` .Net Framework namespace. You do not need to master

threading to write most .Net applications, but the more you know, the better. Thread management is a topic to which data professionals can relate. Think database *transactions*.

A database transaction would not allow two account owners of a joint banking account to withdraw all funds from the bank account just because they execute a withdrawal request at precisely the same time. A database transaction *blocks* one withdrawal request until the other withdrawal request completes, making the second withdrawal request *wait* until the first withdrawal request completes. When the first withdrawal request completes its transaction, the database engine notifies – or *signals* – the second withdrawal request that the blocking process is complete. The second withdrawal request executes and fails because the account is empty – the first withdrawal request already removed the funds from the account while blocking the second withdrawal request.

The terms emphasized in the previous paragraph relate to both data transactions and thread safety.

An application is considered *threadsafe* if the application includes state management functionality similar database transactions. A threadsafe application manages thread safety like a database transaction maintains atomicity as a mechanism for protecting code execution and delivering accurate results.

A threadsafe application achieves atomicity by allowing some code to only execute serially while managing queues of waiting threads. Waiting threads receive signals when their turn to execute arrives. Threadsafe applications are the heart of multi-threaded functionality. Learn more at `docs.microsoft.com/en-us/dotnet/api/system.threading?view=netframework-4.7.2`.

Multi-threaded applications share *resources* – chunks of functionality which are coded for *exclusive* access, like a `WithdrawFunds()` method in a banking application. It is common for a multi-threaded application to manage more than one exclusive resource to complete execution. A `WaitHandle` manages a collection of resources executing in a related operation, and the `WaitAny` method waits for a signal to *any* member of the collection in the `WaitHandle`. Learn more about the `WaitHandle.WaitAny` method at `docs.microsoft.com/en-us/dotnet/api/system.threading.waithandle.waitany?view=netframework-4.7.2`.

A `ManualResetEvent` object is derived from the `WaitHandle` object. The state of a `ManualResetEvent` object is a Boolean value: false or true; and this state is managed by calling the `ManualResetEvent.Set()` and `ManualResetEvent.Reset()`

methods. The `ManualResetEvent.Reset()` method *resets* (sets the Boolean state to *false*), *manually* blocking any additional threads that show up to use the shared resource, letting these threads know they have to wait for their turn to execute. The `ManualResetEvent.Set()` method *signals* (sets the Boolean state to *true*) waiting threads, *manually* letting waiting threads know they may now execute.

> The previous two sentences are important. You may need to come back to this section as we work through the examples to come. Don't feel bad. Threading is hard. Thread safety is harder.

ExecuteCatalogPackageTask.Execute

The SSIS Catalog `Synchronized` execution parameter – A Boolean value – controls how the `ExecuteCatalogPackageTask.Execute` method returns from execution. The default `Synchronized` execution parameter value is false, which means the `ExecuteCatalogPackageTask.Execute` method functions as "fire and forget." If you start an SSIS package with the `Synchronized` execution parameter value set to false, the package starts executing, and the `ExecuteCatalogPackageTask.Execute` method returns "success" almost immediately. Because the `ExecuteCatalogPackageTask.Execute` method behaves as "fire and forget" and returns "success" almost immediately, the `ExecuteCatalogPackageTask.Execute` method's default 30-second timeout does not affect SSIS packages executed with the `Synchronized` execution parameter value set to false.

The 30-second timeout for the `ExecuteCatalogPackageTask.Execute` method is only an issue when an SSIS package is executed with the `Synchronized` execution parameter value set to true.

The implementation of the `WaitHandle.WaitAny`-based solution, then, applies only to SSIS package executions when the `Synchronized` execution parameter value set to true.

Designing a Test SSIS Package

To test 30-second timeout errors for the `ExecuteCatalogPackageTask.Execute` method, add a new test SSIS package named RunForSomeTime.dtsx to a test SSIS project, as shown in Figure 19-1:

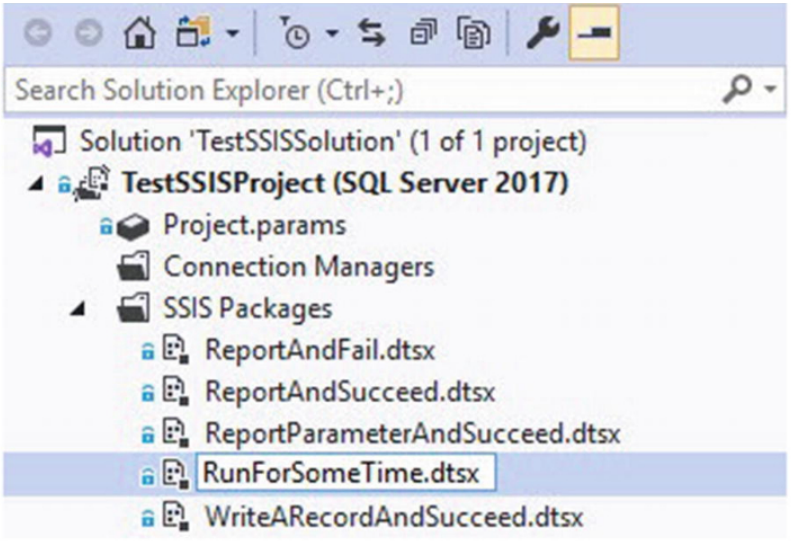


Figure 19-1 Adding the RunForSomeTime.dtsx test SSIS package

Add a package parameter named DelayString of the String data type as shown in Figure 19-2:

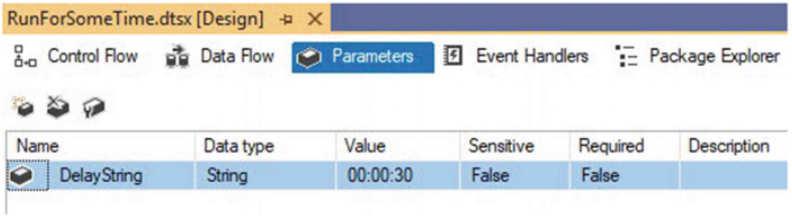


Figure 19-2 Adding the DelayString package parameter

Add an SSIS variable named WaitForQuery of the String data type. Set the value of the WaitForQuery using the T-SQL statement shown in Listing 19-1:

```
WaitFor Delay '00:00:30'
```

Listing 19-1 Adding the WaitForQuery SSIS variable value

Once the T-SQL statement is added to the variable value, the WaitForQuery variable appears as shown in Figure 19-3:

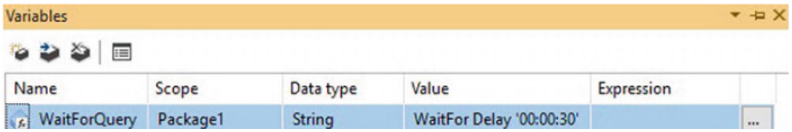


Figure 19-3 Adding the WaitForQuery SSIS variable

Click the ellipsis beside the Expression textbox to open the Expression Builder dialog, and then enter the SSIS Expression Language statement in Listing 19-2 in the Expression textbox:

```
"WaitFor Delay '" + @[$Package::DelayString]  + "'"
```

Listing 19-2 the WaitForQuery SSIS variable expression

When added, the WaitForQuery SSIS variable expression appears as shown in Figure 19-4:

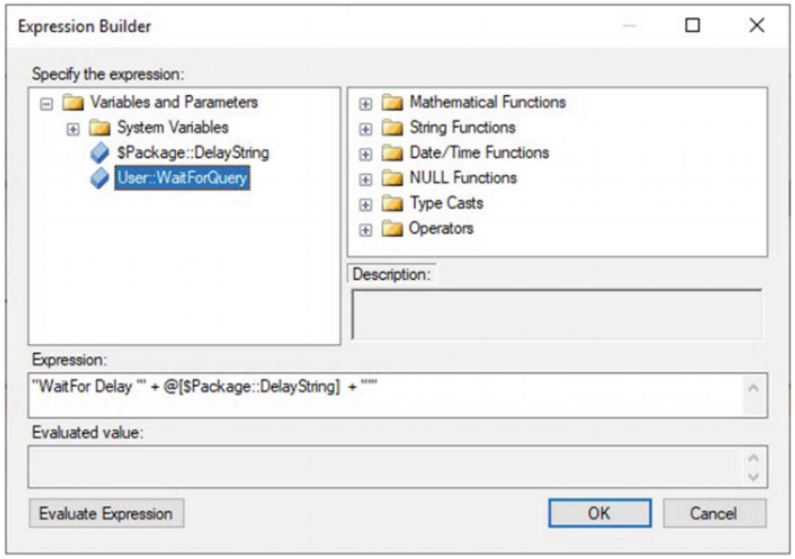
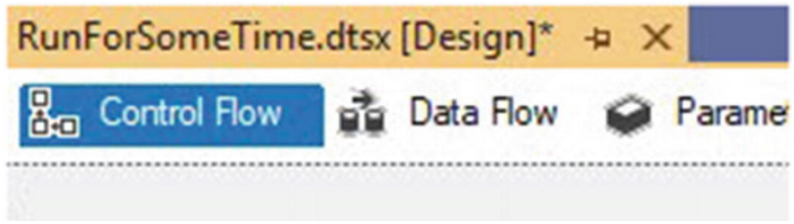


Figure 19-4 The WaitForQuery SSIS variable expression

Click the OK button to close the Expression Builder dialog.

Add an Execute SQL Task to RunForSomeTime.dtsx and rename the execute SQL task "SQL Run for some time," as shown in Figure 19-5:



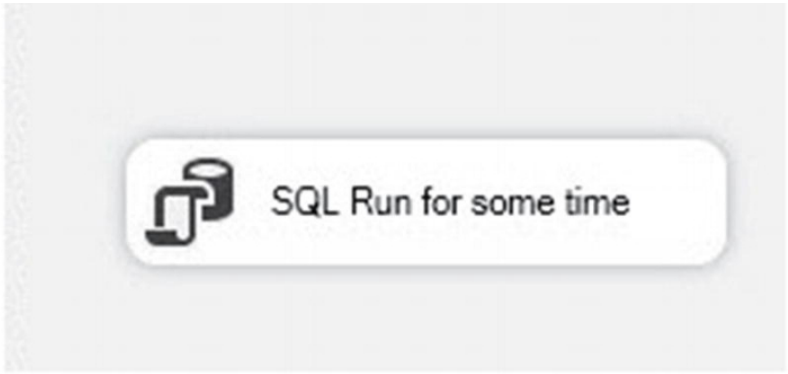


Figure 19-5 Adding the SQL Run for some time Execute SQL Task

Open the SQL Run for some time execute SQL task editor and set the ConnectionType to ADO.NET. Click the Connection dropdown and click "<New connection…>." Configure an ADO.Net connection manager to *any* database – this package will not use the configured connection, but the Execute SQL Task requires a connection manager configured.

Set the SQLSourceType property to "Variable." Set the SourceVariable property to the `User::WaitForQery` SSIS variable, as shown in Figure 19-6:

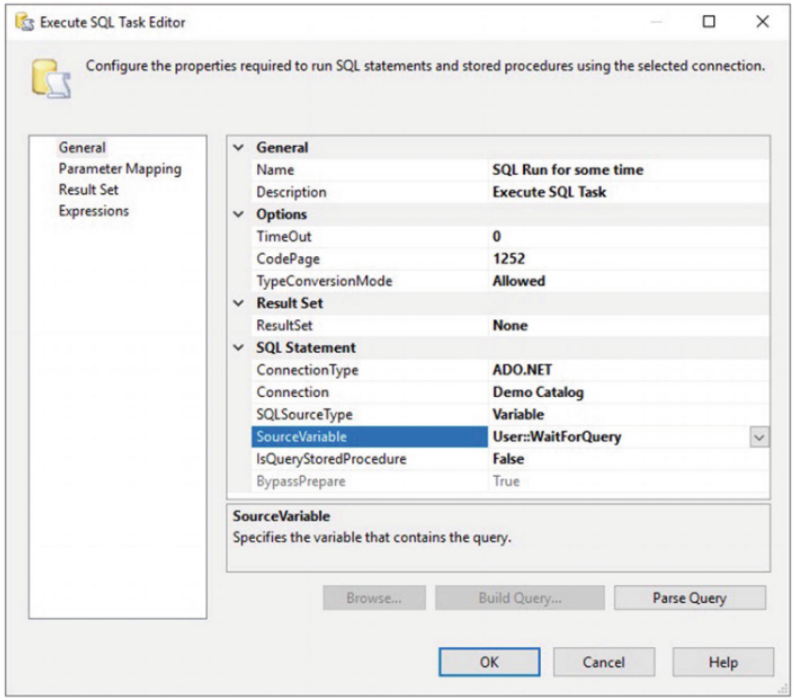


Figure 19-6 Configuring SQL Run for some time execute SQL task

Click the OK button to close the execute SQL task editor, and then deploy the SSIS project to an SSIS Catalog.

Next, open SSMS and connect to the SSIS Catalog to which you deployed the SSIS project. Navigate to the SSIS project in the SSMS Object Explorer Integration Services Catalogs node, right-click the project, and then click Configure, as shown in Figure 19-7:

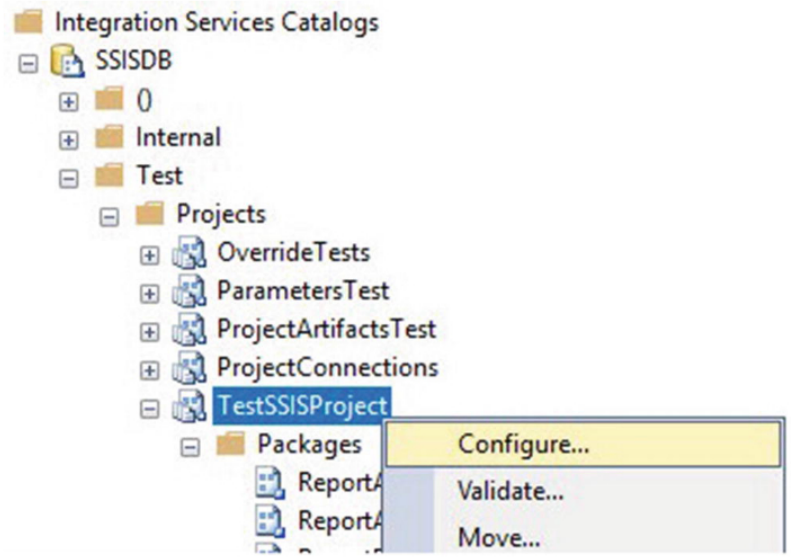


Figure 19-7 Configure the SSIS project

When the Configure dialog displays, click the ellipsis beside the RunForSomeTime.dtsx SSIS package's DelayString parameter value, as shown in Figure 19-8:

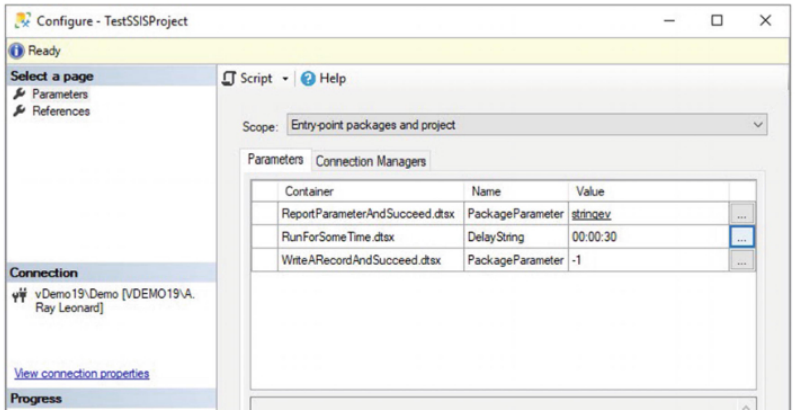


Figure 19-8 Preparing to override the RunForSomeTime.dtsx SSIS package DelayString parameter

Configure a literal override of the RunForSomeTime.dtsx SSIS package DelayString parameter by clicking the "Edit value" option and entering "00:00:45" in the literal override textbox, as shown in Figure 19-9:

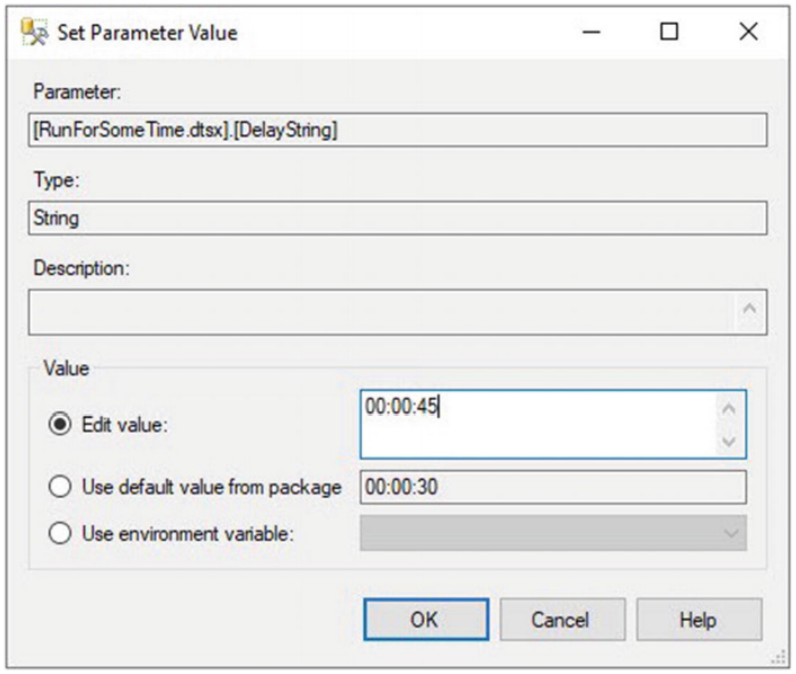
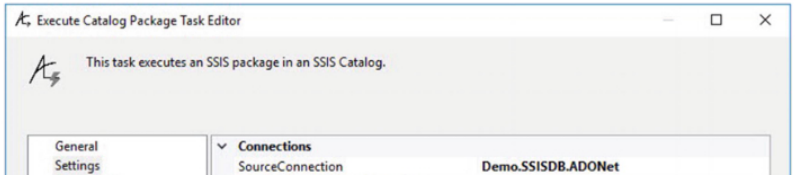


Figure 19-9 Configure a literal override of the RunForSomeTime.dtsx SSIS package DelayString parameter

Click the OK button to save the literal override.

Open a test SSIS package, add an Execute Catalog Package Task, and configure the Execute Catalog Package Task to execute the RunForSomeTime.dtsx SSIS package. Be sure to leave the `Synchronized` property set to false, as shown in Figure 19-10:



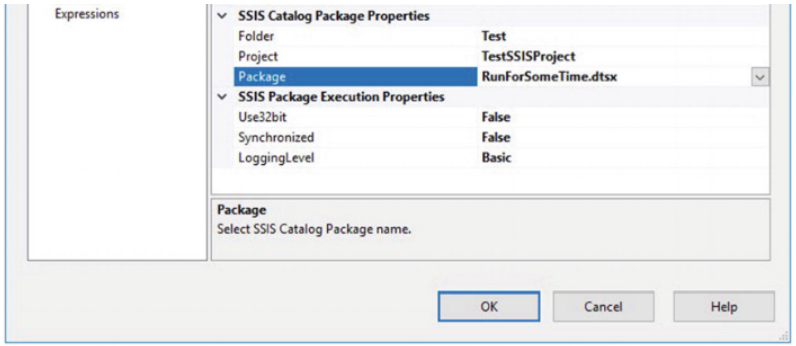


Figure 19-10 Executing the RunForSomeTime.dtsx SSIS package

Execute the test SSIS package and review the Progress tab when execution completes (and succeeds, if all goes as planned), as shown in Figure 19-11:

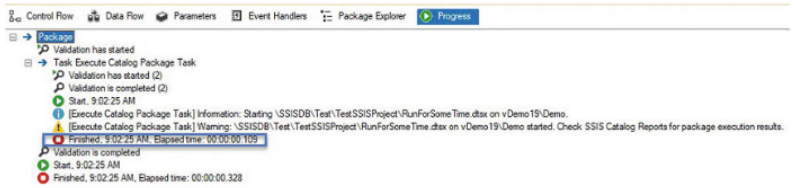


Figure 19-11 Reviewing the Progress tab

Note the Execute Catalog Package Task reports finished after less than one second.

Open the SSIS Catalog Reports All Executions report in SSMS, as shown in Figure 19-12:

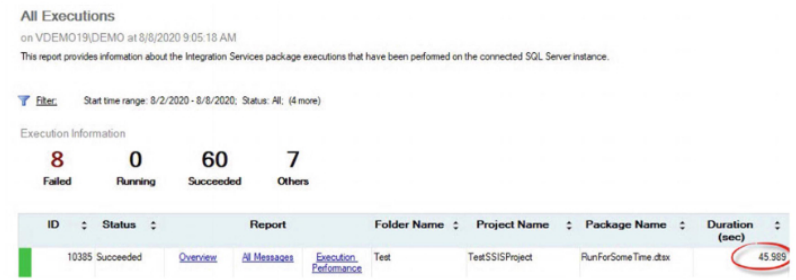


Figure 19-12 Viewing the All Executions SSIS Catalog report

Note the RunForSomeTime.dtsx SSIS package executed for just over 45 seconds, which is how it was configured earlier.

Return to the test SSIS package's Execute Catalog Package Task editor, and set the Synchronized property to True, as shown in Figure 19-13:

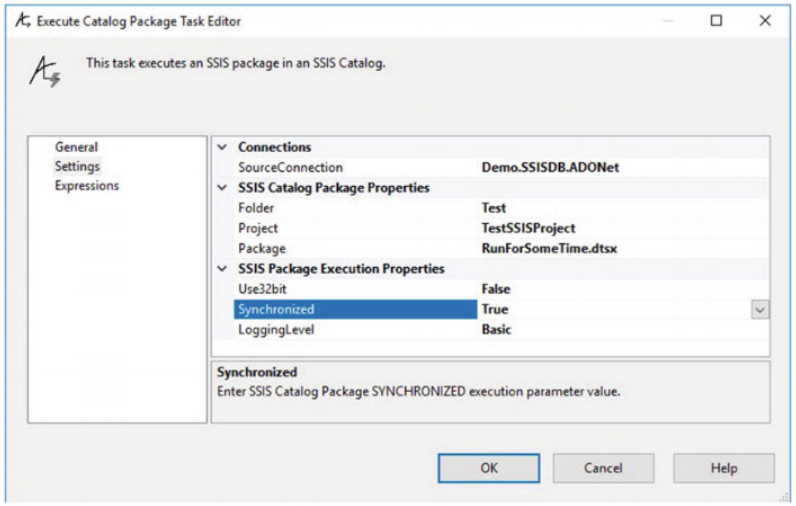


Figure 19-13 Setting the Synchronized property to true

Execute the test SSIS package. After 30 seconds, note the test SSIS package execution fails with the error message "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.)," as shown in Figure 19-14:

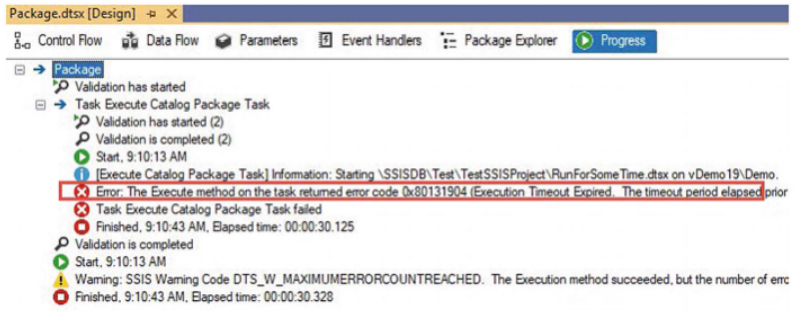


Figure 19-14 Execution timeout at 30 seconds

This is the behavior we wish to overcome. The RunForSomeTime.dtsx SSIS package may now be used to test the implementation of the `WaitHandle.WaitAny`-based solution.

Implementing a WaitHandle.WaitAny Method

The heart of this `WaitHandle.WaitAny`-based solution for executing SSIS packages that take more than 30 seconds to execute with the `Synchronized` property

set to true is to execute the SSIS package *asynchronously*. Executing the SSIS package asynchronously is, well, executing the SSIS package as if the `Synchronized` property set to *false*, so this method of executing the SSIS package will be invoked if and only if the `Synchronized` property is set to *true*.

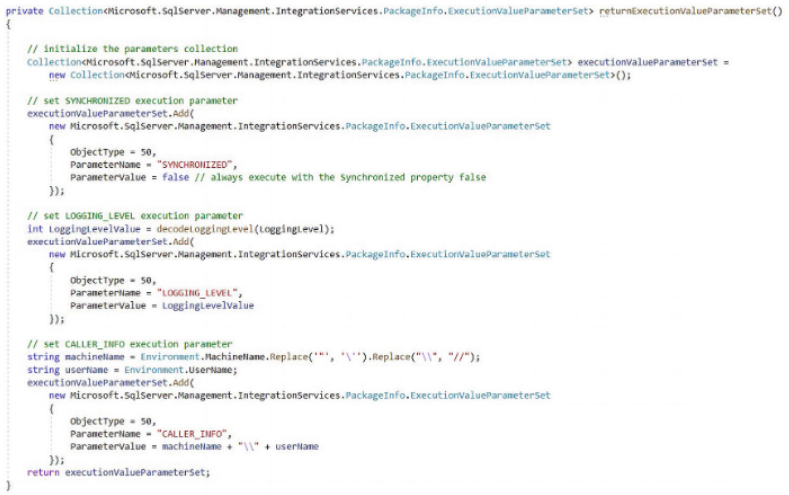
Begin implementing this `WaitHandle.WaitAny`-based solution by editing the `returnExecutionValueParameterSet` method in the `ExecuteCatalogPackageTask` class. While editing, add code to set the `CALLER_INFO` execution parameter using the code in Listing 19-3:

```
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
  {
    ObjectType = 50,
    ParameterName = "SYNCHRONIZED",
    ParameterValue = false // always execute with the Synchronized prope
  });
  // set LOGGING_LEVEL execution parameter
  int LoggingLevelValue = decodeLoggingLevel(LoggingLevel);
  executionValueParameterSet.Add(new Microsoft.SqlServer.Management.Inte
  {
    ObjectType = 50,
    ParameterName = "LOGGING_LEVEL",
    ParameterValue = LoggingLevelValue
  });
  // set CALLER_INFO execution parameter
  string machineName = Environment.MachineName.Replace('"', '\'').Replac
  string userName = Environment.UserName;
  executionValueParameterSet.Add(new Microsoft.SqlServer.Management.Inte
  {
    ObjectType = 50,
    ParameterName = "CALLER_INFO",
    ParameterValue = machineName + "\\" + userName
  });
  return executionValueParameterSet;
}
```

Listing 19-3 Editing the returnExecutionValueParameterSet method in ExecuteCatal

Once edited, the `returnExecutionValueParameterSet` method appears as shown in Figure 19-15:



```
private Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet> returnExecutionValueParameterSet()
{
    // initialize the parameters collection
    Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet> executionValueParameterSet =
        new Collection<Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet>();

    // set SYNCHRONIZED execution parameter
    executionValueParameterSet.Add(
        new Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet
        {
            ObjectType = 50,
            ParameterName = "SYNCHRONIZED",
            ParameterValue = false // always execute with the Synchronized property false
        });

    // set LOGGING_LEVEL execution parameter
    int LoggingLevelValue = decodeLoggingLevel(LoggingLevel);
    executionValueParameterSet.Add(
        new Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet
        {
            ObjectType = 50,
            ParameterName = "LOGGING_LEVEL",
            ParameterValue = LoggingLevelValue
        });

    // set CALLER_INFO execution parameter
    string machineName = Environment.MachineName.Replace('"', '\'').Replace("\\", "//");
    string userName = Environment.UserName;
    executionValueParameterSet.Add(
        new Microsoft.SqlServer.Management.IntegrationServices.PackageInfo.ExecutionValueParameterSet
        {
            ObjectType = 50,
            ParameterName = "CALLER_INFO",
            ParameterValue = machineName + "\\" + userName
        });
    return executionValueParameterSet;
}
```

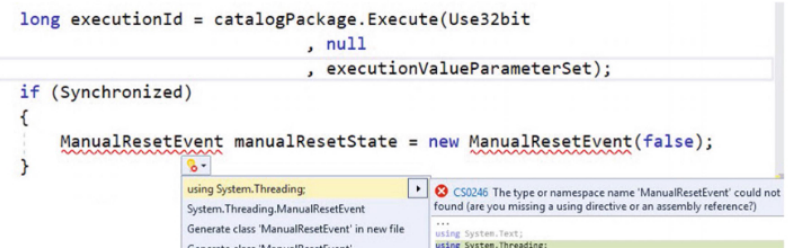
Figure 19-15 Editing the returnExecutionValueParameterSet method

Continue implementing this `WaitHandle.WaitAny`-based solution by editing the `ExecuteCatalogPackageTask.Execute` method `catalogPackage.Execute` call by editing the call to `catalogPackage.Execute`, adding the code in Listing 19-4:

```
long executionId = catalogPackage.Execute(Use32bit
                                          , null
                                          , executionValueParameterSet);
if (Synchronized)
{
  ManualResetEvent manualResetState = new ManualResetEvent(false);
}
```

Listing 19-4 Checking for execution configured for Synchronized execution

When added, the code appears as shown in Figure 19-16:



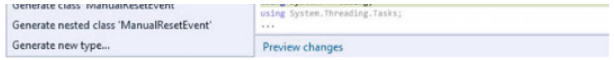


Figure 19-16 Implementing the WaitHandle.WaitAny-based solution

Use Visual Studio Quick Actions to add the `using System.Threading;` directive by hovering over the ManualResetEvent type declaration, clicking the Quick Actions dropdown, and then clicking the `using System.Threading;` directive. After the `using System.Threading;` directive is added, the code appears as shown in Figure 19-17:

```
long executionId = catalogPackage.Execute(Use32bit
                                , null
                                , executionValueParameterSet);
if (Synchronized)
{
    ManualResetEvent manualResetState = new ManualResetEvent(false);
}
```

Figure 19-17 Error cleared after adding the directive

If the package execution is *not* configured for Synchronized execution, the code executes the package as before. If the package execution *is* configured for Synchronized execution, a `ManualResetEvent` named `manualResetState` is declared, and the state of `manualResetState` is initialized as false. Remember, a `ManualResetEvent` set to false begins *waiting* (blocking) any other threads that show up to use the shared resource assigned to the `ManualResetEvent`.

The next step is to configure the `Timer`, which *ticks* based on interval configuration. When the timer ticks, the timer will call a method named `CheckStatus` in a class named `CheckWaitState`. Add the `CheckWaitState` class to the ExecuteCatalogPackageTask.cs file and declare private members using the code in Listing 19-5:

```
class CheckWaitState
{
  private long executionId;
  private int invokeCount;
  private ManualResetEvent manualResetState;
  private int maximumCount;
  private string catalogName;
  private SqlConnection connection;
  private ExecuteCatalogPackageTask task;
  public Operation.ServerOperationStatus OperationStatus { get; set; }
}
```

Listing 19-5 Implementing CheckWaitState in the ExecuteCatalogPackageTask.cs file

Once the code is added, it appears as shown in Figure 19-18:

```
class CheckWaitState
{
    private long executionId;
    private int invokeCount;
    private ManualResetEvent manualResetState;
    private int maximumCount;
    private string catalogName;
    private SqlConnection connection;
    private ExecuteCatalogPackageTask task;
    0 references | Andy Leonard, 8 days ago | 1 author, 4 changes
    public Operation.ServerOperationStatus OperationStatus { get; set; }
}
```

Figure 19-18 Implementing the CheckWaitState class

Implement the CheckWaitState constructor using the code in Listing 19-6:

```
public CheckWaitState(long executionId
   , int maxCount
   , SqlConnection connection
   , string catalogName
   , ExecuteCatalogPackageTask task)
{
  this.executionId = executionId;
  this.invokeCount = 0;
  this.maximumCount = maxCount;
  this.connection = connection;
  this.catalogName = catalogName;
  this.task = task;
}
```

Listing 19-6 Implementing the CheckWaitState constructor

Once the constructor is implemented, the CheckWaitState class appears as shown in Figure 19-19:

```
class CheckWaitState
{
    private long executionId;
    private int invokeCount;
    private ManualResetEvent manualResetState;
    private int maximumCount;
    private string catalogName;
    private SqlConnection connection;
    private ExecuteCatalogPackageTask task;
    0 references | Andy Leonard, 8 days ago | 1 author, 4 changes
    public Operation.ServerOperationStatus OperationStatus { get; set; }


    0 references | Andy Leonard, 8 days ago | 1 author, 2 changes
    public CheckWaitState(long executionId
                        , int maxCount
```

```
                    , SqlConnection connection
                    , string catalogName
                    , ExecuteCatalogPackageTask task)
    {
        this.executionId = executionId;
        this.invokeCount = 0;
        this.maximumCount = maxCount;
        this.connection = connection;
        this.catalogName = catalogName;
        this.task = task;
    }
}
```

Figure 19-19 The CheckWaitState class with the constructor implemented

Implement the `CheckTimerState` class using the code in Listing 19-7:

```
class TimerCheckerState
{
  public ManualResetEvent manualResetState { get; private set; }
  public TimerCheckerState(ManualResetEvent manualResetState)
  {
    this.manualResetState = manualResetState;
  }
}
```

Listing 19-7 Implementing the TimerCheckerState class

Once added, the code appears as shown in Figure 19-20:

```
class TimerCheckerState
{
    2 references
    public ManualResetEvent manualResetState { get; private set; }

    1 reference
    public TimerCheckerState(ManualResetEvent manualResetState)
    {
        this.manualResetState = manualResetState;
    }
}
```

Figure 19-20 The TimerCheckerState class

The next step is to implement a helper function named `returnOperationStatus` in the `CheckWaitState` class using the code in Listing 19-8:

```
public Operation.ServerOperationStatus returnOperationStatus()
{
  CatalogCollection catalogCollection = new IntegrationServices(connecti
```

```
    Catalog catalog = catalogCollection[catalogName];
    OperationCollection operationCollection = catalog.Operations;
    Operation operation = operationCollection[executionId];
    Operation.ServerOperationStatus operationStatus = operation.Status;
    return operationStatus;
  }
```

Listing 19-8 Coding the CheckWaitState.returnOperationStatus helper function

Once added to the `CheckWaitState` class, the `returnOperationStatus` helper function appears as shown in Figure 19-21:

```
public Operation.ServerOperationStatus returnOperationStatus()
{
    CatalogCollection catalogCollection = new IntegrationServices(connection).Catalogs;
    Catalog catalog = catalogCollection[catalogName];
    OperationCollection operationCollection = catalog.Operations;
    Operation operation = operationCollection[executionId];
    Operation.ServerOperationStatus operationStatus = operation.Status;

    return operationStatus;
}
```

Figure 19-21 The CheckWaitState.returnOperationStatus helper function

The returnOperationStatus method first connects to the `CatalogCollection` type named `catalogCollection`. A `Catalog` type named `catalog` is derived from the `catalogCollection` by the `catalogName` property (defaulted to "SSISDB").

An `OperationCollection` type named `operationCollection` is assigned from the `catalog.Operations` type collection. An `Operation` type named `operation` is read from the `operationCollection[executionId]`. The `operationStatus` variable, a variable of the `Operation.ServerOperationStatus` type, is read from the `operation.Status` property.

The next step is to code the `checkOperationStatus` method to the CheckWaitState class to respond to the `ServerOperationStatus` of the SSIS package execution using the code in Listing 19-9:

```
public void checkOperationStatus(Operation.ServerOperationStatus operati
{
  // check for package execution "finished" states
  if (
      (operationStatus == Operation.ServerOperationStatus.Canceled)
   || (operationStatus == Operation.ServerOperationStatus.Completion)
   || (operationStatus == Operation.ServerOperationStatus.Failed)
   || (operationStatus == Operation.ServerOperationStatus.Stopping)
   || (operationStatus == Operation.ServerOperationStatus.Success)
```

```
    || (operationStatus == Operation.ServerOperationStatus.UnexpectTermin
      )
   {


     // reset counter
     invokeCount = 0;
     // signal thread
     manualResetState.Set();
   }
 }
```

Listing 19-9 Add the CheckWaitState.CheckOperationStatus method

Once added, the code appears as shown in Figure 19-22:

```
public void checkOperationStatus(Operation.ServerOperationStatus operationStatus)
{
    // check for package execution "finished" states
    if (
            (operationStatus == Operation.ServerOperationStatus.Canceled)
        || (operationStatus == Operation.ServerOperationStatus.Completion)
        || (operationStatus == Operation.ServerOperationStatus.Failed)
        || (operationStatus == Operation.ServerOperationStatus.Stopping)
        || (operationStatus == Operation.ServerOperationStatus.Success)
        || (operationStatus == Operation.ServerOperationStatus.UnexpectTerminated)
      )
    {
        // reset counter
        invokeCount = 0;
        // signal thread
        manualResetState.Set();
    }
}
```

Figure 19-22 Adding the CheckWaitState.CheckOperationStatus method

The `checkOperationStatus` method receives an `Operation.ServerOperationStatus` type argument named `operationStatus`. An `if` conditional checks to see if the `operationStatus` argument value is in a "finished" operational state. "Finished" operation states include

- Canceled
- Completion
- Failed
- Stopping
- Success
- UnexpectTerminated

If the `operationStatus` argument value is in a "finished" operational state, the timer tick counter named `invokeCount` is reset to 0, and the `ManualResetEvent`

type property named `manualResetState` is `Set`. As stated earlier: The `ManualResetEvent.Set()` method *signals* (sets the Boolean state to *true*) waiting threads, *manually* letting waiting threads know they may now execute.

Refactoring logMessage to raiseEvent

Return to the `ExecuteCatalogPackageTask` class and refactor the `logMessage` method , starting with the name of the method. The `logMessage` method doesn't actually log a message. Instead, the `logMessage` method raises an event. Refactor the `logMessage` method by renaming it to `raiseEvent` and replacing it with the code in Listing **19-10**:

```
public void raiseEvent(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;
  }
}
```