Resolving Unsupported Programmable Objects

When importing programmable objects into your SQL Change Automation project for the first time, you may notice that one or more scripts are placed into a folder called *Unsupported*, instead of the usual *Programmable Objects* folder. This can happen to a script containing a stored procedure/view/user defined function object if one or more of the following applies:

- Scenario A: The object contains the SCHEMABINDING clause and it is part of a dependency chain with other schema-bound objects
- Scenario B: *(User Defined Functions only)* There is a dependency on the object from a table object, for example a computed column or check constraint with a call to the function
- Scenario C: *(User Defined Functions in SQL Server 2016+/Azure SQL Database only)* There is a dependency on the function from a security policy object
- Scenario D: The object contains the NATIVE_COMPILATION clause, preventing it from being deployed within a user transaction

If any of the above applies, SQL Change Automation will separate the affected object files into a folder called *Unsupported*, which is not included in project build or database deployment operations. The files are moved into this folder during the import process to prevent the scripts from failing at deployment or verification time.

Introduction to "dependency chain" scenarios

By default, SQL Change Automation will import all of your schema objects -- both table and programmable object types -- into numerically ordered migration scripts. However, you can decide to split programmable object types (stored procedures, views, functions etc) into individual files to simplify source control branch and merge activities, allowing your team to make changes to code objects in parallel.

This works well for objects with "soft" dependencies: for example, a view that selects data from another view, or a stored procedure that calls another stored procedure, since those types of objects can be repeatedly dropped and recreated in any order without generating errors at deployment time. Things get more complicated when dealing with "hard" dependencies, such as when a schema-bound view depends on another schema-bound view, or when a table contains a check constraint that depends on a user-defined function.

To preserve the integrity of schema-bound objects, SQL Server enforces the dependencies between such objects at deployment time. If you try to change any object that is chained to another object without first unwinding the dependency tree, SQL Server will prevent the change from occurring by raising an error, causing the deployment to be aborted.

Scenario A: Interdependent schema-bound objects

To demonstrate how objects in a dependency chain can be turned into programmable object scripts, let's take a scenario that involves two schema-bound views:

```
-- ViewA.sql
CREATE VIEW [dbo].[ViewA]
WITH SCHEMABINDING
AS
SELECT 1 As MyCol;
GO

-- ViewB.sql
CREATE VIEW [dbo].[ViewB]
WITH SCHEMABINDING
AS
SELECT MyCol FROM [dbo].[ViewA];
GO
```

Here [ViewB] has a dependency on [ViewA]. If we try to make a change to [ViewA], e.g.

```
ALTER VIEW [dbo].[ViewA]
WITH SCHEMABINDING
AS
SELECT 2 As MyCol;
GO
```

The following error will be raised:

```
Msg 3729, Level 16, State 3, Procedure ViewA, Line 16
Cannot ALTER 'dbo.ViewA' because it is being referenced by object 'ViewB'.
```

SQL Server has indicated that the change cannot be made until all references to the object are removed. The difficulty with using SQL Change Automation's programmable objects feature in this scenario is that every script file is executed independently of the others, so dropping and recreating all the objects in the dependency tree is not possible.

In order to turn the *unsupported* files into programmable object files, the objects must be combined into a single script to allow the entire dependency tree to be deployed in one atomic operation.

In summary, the programmable object script must perform the following operations:

1. Drop the child object(s), e.g. [ViewB]
2. Drop the parent object, e.g. [ViewA]
3. Create the parent object, e.g. [ViewA]
4. Create the child object(s), e.g. [ViewB]

The programmable object script (Programmable Objects\Views\dbo.ViewA_dbo.ViewB.sql) will end up looking something like this:

Scenario A - Programmable Object script

```
IF OBJECT_ID('[dbo].[ViewB]') IS NOT NULL
DROP VIEW [dbo].[ViewB];
GO

IF OBJECT_ID('[dbo].[ViewA]') IS NOT NULL
DROP VIEW [dbo].[ViewA];
GO

CREATE VIEW [dbo].[ViewA]
WITH SCHEMABINDING
AS
SELECT 1 As MyCol;
GO

CREATE VIEW [dbo].[ViewB]
WITH SCHEMABINDING
AS
SELECT MyCol FROM [dbo].[ViewA];
GO
```

In this example, both views are dropped and recreated in order of dependency. The conditional logic around the DROP statements ensures that the script is re-runnable (idempotent), allowing changes to the objects to deployed incrementally to your target databases.

Scenario B: Tables with dependencies on Functions

To demonstrate how a table that has a dependency on a function can be turned into a programmable object script, let's take the scenario of a computed column that includes a call to a user defined function. For example, the *AdventureWorks2014* sample database contains the following table definition:

```
CREATE TABLE [Sales].[Customer]
(
[CustomerID] [int] NOT NULL IDENTITY(1, 1),
[PersonID] [int] NULL,
[StoreID] [int] NULL,
[TerritoryID] [int] NULL,
[AccountNumber] AS (isnull('AW'+[dbo].[ufnLeadingZeros]([CustomerID]),'')),
[rowguid] [uniqueidentifier] NOT NULL ROWGUIDCOL CONSTRAINT [DF_Customer_rowguid] DEFAULT (newid()),
[ModifiedDate] [datetime] NOT NULL CONSTRAINT [DF_Customer_ModifiedDate] DEFAULT (getdate())
)
ALTER TABLE [Sales].[Customer] ADD CONSTRAINT [PK_Customer_CustomerID] PRIMARY KEY CLUSTERED  ([CustomerID])
GO
CREATE UNIQUE NONCLUSTERED INDEX [AK_Customer_AccountNumber] ON [Sales].[Customer] ([AccountNumber])
GO
```

The table's [AccountNumber] column has a reference to the function object, [ufnLeadingZeros]:

```
SET QUOTED_IDENTIFIER ON
SET ANSI_NULLS ON
GO
CREATE FUNCTION [dbo].[ufnLeadingZeros](
    @Value int
)
RETURNS varchar(8)
WITH SCHEMABINDING
AS
BEGIN
    DECLARE @ReturnValue varchar(8);

    SET @ReturnValue = CONVERT(varchar(8), @Value);
    SET @ReturnValue = REPLICATE('0', 8 - DATALENGTH(@ReturnValue)) + @ReturnValue;

    RETURN (@ReturnValue);
END;
GO
```

Note that the computed column also has an index on it. In order to make it possible to change the function in an idempotent way, the dependency tree that involves the computed column and that index upon that column must be unwound and then recreated in the appropriate order within the programmable object script itself:

Scenario B - Programmable Object script

```
IF OBJECT_ID('[dbo].[ufnLeadingZeros]') IS NOT NULL
BEGIN
        DROP INDEX [Sales].[Customer].[AK_Customer_AccountNumber];
        ALTER TABLE [Sales].[Customer] DROP COLUMN [AccountNumber];
        DROP FUNCTION [dbo].[ufnLeadingZeros];
END
GO
SET ANSI_NULLS ON
SET QUOTED_IDENTIFIER ON
GO

CREATE FUNCTION [dbo].[ufnLeadingZeros](
    @Value int
)
RETURNS varchar(8)
WITH SCHEMABINDING
AS
BEGIN
    DECLARE @ReturnValue varchar(8);

    SET @ReturnValue = CONVERT(varchar(8), @Value);
    SET @ReturnValue = REPLICATE('0', 8 - DATALENGTH(@ReturnValue)) + @ReturnValue;

    RETURN (@ReturnValue);
END;
GO

ALTER TABLE [Sales].[Customer]
        ADD [AccountNumber] AS (isnull('AW'+[dbo].[ufnLeadingZeros]([CustomerID]),''));
GO

CREATE UNIQUE NONCLUSTERED INDEX [AK_Customer_AccountNumber]
        ON [Sales].[Customer] ([AccountNumber]);
GO
```

Changes to both the function, as well as the computed column that uses the function, can then be made by simply editing the programmable object file and deploying the SQL Change Automation project.

Scenario C: Security Policies with dependencies on Functions

To demonstrate how a security policy that has a dependency on a function can be turned into a programmable object script, let's take the following example from the *WideWorldImporters* sample database:

```
CREATE SECURITY POLICY [Application].[FilterCustomersBySalesTerritoryRole]
ADD FILTER PREDICATE [Application].[DetermineCustomerAccess]([DeliveryCityID])
ON [Sales].[Customers],
ADD BLOCK PREDICATE [Application].[DetermineCustomerAccess]([DeliveryCityID])
ON [Sales].[Customers] AFTER UPDATE
WITH (STATE = ON)
GO
```

The security policy references the [Application].[DetermineCustomerAccess] function:

```
SET QUOTED_IDENTIFIER ON
SET ANSI_NULLS ON
GO

CREATE FUNCTION [Application].[DetermineCustomerAccess](@CityID int)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN (SELECT 1 AS AccessResult
        WHERE IS_ROLEMEMBER(N'db_owner') <> 0
        OR IS_ROLEMEMBER((SELECT sp.SalesTerritory
                          FROM [Application].Cities AS c
                          INNER JOIN [Application].StateProvinces AS sp
                          ON c.StateProvinceID = sp.StateProvinceID
                          WHERE c.CityID = @CityID) + N' Sales') <> 0
            OR (ORIGINAL_LOGIN() = N'Website'
                    AND EXISTS (SELECT 1
                                FROM [Application].Cities AS c
                                        INNER JOIN [Application].StateProvinces AS sp
                                        ON c.StateProvinceID = sp.StateProvinceID
                                        WHERE c.CityID = @CityID
                                        AND sp.SalesTerritory = SESSION_CONTEXT(N'SalesTerritory'))));
GO
```

In order to make it possible to change the function within a programmable object script, the security policy must first be dropped and then recreated after the function is modified:

Scenario C - Programmable Object script

```
IF OBJECT_ID('[Application].[DetermineCustomerAccess]') IS NOT NULL
BEGIN
        DROP SECURITY POLICY [Application].[FilterCustomersBySalesTerritoryRole];
        DROP FUNCTION [Application].[DetermineCustomerAccess];
END

SET QUOTED_IDENTIFIER ON
SET ANSI_NULLS ON
GO

CREATE FUNCTION [Application].[DetermineCustomerAccess](@CityID int)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN (SELECT 1 AS AccessResult
        WHERE IS_ROLEMEMBER(N'db_owner') <> 0
        OR IS_ROLEMEMBER((SELECT sp.SalesTerritory
                          FROM [Application].Cities AS c
                          INNER JOIN [Application].StateProvinces AS sp
                          ON c.StateProvinceID = sp.StateProvinceID
                          WHERE c.CityID = @CityID) + N' Sales') <> 0
            OR (ORIGINAL_LOGIN() = N'Website'
                    AND EXISTS (SELECT 1
                                FROM [Application].Cities AS c
                                        INNER JOIN [Application].StateProvinces AS sp
                                        ON c.StateProvinceID = sp.StateProvinceID
                                        WHERE c.CityID = @CityID
                                        AND sp.SalesTerritory = SESSION_CONTEXT(N'SalesTerritory'))));

GO

CREATE SECURITY POLICY [Application].[FilterCustomersBySalesTerritoryRole]
ADD FILTER PREDICATE [Application].[DetermineCustomerAccess]([DeliveryCityID])
ON [Sales].[Customers],
ADD BLOCK PREDICATE [Application].[DetermineCustomerAccess]([DeliveryCityID])
ON [Sales].[Customers] AFTER UPDATE
WITH (STATE = ON)
GO
```

Changes to both the function, as well as the security policy that uses the function, can then be made by simply editing the programmable object file and deploying the SQL Change Automation project.

Scenario D: Deploying natively-compiled objects

Objects that contain the NATIVE_COMPILATION clause require special handling because it is not possible to create, drop or alter these types of objects within a user transaction. This is important because typically SQL Change Automation will try to execute all of your migrations and programmable objects within a single transaction to ensure that the deployment is performed atomically. However, in order to deploy natively-compiled stored procedures and functions, it is necessary to disable SQL Change Automation's automatic transaction handling. This can be done at the script level by pasting the following metadata onto the first line of affected script(s):

```
-- <Migration TransactionHandling="Custom" />
GO
```

For example, this is how the [RecordColdRoomTemperatures] procedure in the *WideWorldImporters* sample database could be turned into a programmable object:

Scenario D - Programmable Object script

```
-- <Migration TransactionHandling="Custom" />
GO
IF OBJECT_ID('[Website].[RecordColdRoomTemperatures]') IS NOT NULL
        DROP PROCEDURE [Website].[RecordColdRoomTemperatures];
GO

SET QUOTED_IDENTIFIER ON
SET ANSI_NULLS ON
GO

CREATE PROCEDURE [Website].[RecordColdRoomTemperatures]
@SensorReadings Website.SensorDataList READONLY
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS
BEGIN ATOMIC WITH
(
        TRANSACTION ISOLATION LEVEL = SNAPSHOT,
        LANGUAGE = N'English'
)
    BEGIN TRY

                DECLARE @NumberOfReadings int = (SELECT MAX(SensorDataListID) FROM @SensorReadings);
                DECLARE @Counter int = (SELECT MIN(SensorDataListID) FROM @SensorReadings);

                DECLARE @ColdRoomSensorNumber int;
                DECLARE @RecordedWhen datetime2(7);
                DECLARE @Temperature decimal(18,2);

                -- note that we cannot use a merge here because multiple readings might exist for each sensor

                WHILE @Counter <= @NumberOfReadings
                BEGIN
                        SELECT @ColdRoomSensorNumber = ColdRoomSensorNumber,
                               @RecordedWhen = RecordedWhen,
                                   @Temperature = Temperature
                        FROM @SensorReadings
                        WHERE SensorDataListID = @Counter;

                        UPDATE Warehouse.ColdRoomTemperatures
                                SET RecordedWhen = @RecordedWhen,
                                    Temperature = @Temperature
                        WHERE ColdRoomSensorNumber = @ColdRoomSensorNumber;

                        IF @@ROWCOUNT = 0
                        BEGIN
                                INSERT Warehouse.ColdRoomTemperatures
                                        (ColdRoomSensorNumber, RecordedWhen, Temperature)
                                VALUES (@ColdRoomSensorNumber, @RecordedWhen, @Temperature);
                        END;

                        SET @Counter += 1;
                END;

    END TRY
    BEGIN CATCH
        THROW 51000, N'Unable to apply the sensor data', 2;

        RETURN 1;
    END CATCH;
END;


GO
```

As it is necessary to first COMMIT any open transactions before deploying natively-compiled objects, and a new transaction opened after it to deploy the remaining objects, it is recommended that that these objects be prioritized last in the deployment order (assuming there are no dependencies on the natively-compiled objects).