Transaction Handling

When deploying your database with ReadyRoll, you may wonder:

- How are transactions handled?
- If the deployment fails, will all the previous migrations be rolled-back?
- Can I implement my own transaction handling?

The answer to these questions depends on the type of change you are performing. Here is a break-down of transaction handling behaviour in ReadyRoll:

Sequence	Script Directory	Transaction Used	Example script filename
1	Pre-Deployment	No	01_Create_Database.sql
2	Migrations	Yes	0010_Invoices_AddColumn_Date.sql
3	Programmable Objects & Additional Scripts	Yes	Views\dbo.vwInvoicesByYear.sql
4	Post-Deployment	No	01_Finalize_Deployment.sql

Where *Transaction Used* = *Yes*, ReadyRoll will automatically wrap your migrations in a single BEGIN TRAN / COMMIT TRAN "block".

If at any point one of these migration fails to deploy, the entire transaction will be rolled-back.

If you wish to use a transaction within your Pre/Post-Deployment scripts, you can perform a BEGIN TRAN yourself, however please ensure that you perform a COMMIT or ROLLBACK at the end of your script to avoid having overlapping transactions.

In the case of migration scripts, programmable objects and additional scripts, ReadyRoll will confirm that *TRANCOUNT() = 0* after each script execution.

Anatomy of Transaction Handling in ReadyRoll

This section details some of the conventions used in ReadyRoll deployments by examining the project artifacts; specifically the T-SQL file that is generated when you perform a *Build* in Visual Studio, eg. `AdventureWorks\bin\Debug\AdventureWorks.sql`. This file contains a concatenated list of all the scripts from your ReadyRoll database project.

Output script header

The first thing you will see when you open the deployment script is the SqlCmd header:

```
----
================================================================================================================================
====
---- SQLCMD Variables

:setvar DatabaseName "AdventureWorks"
:setvar Configuration "Debug"
:setvar OctopusEnvironmentName "DEV"
:setvar ReportFolder "\\bigsan01\reports\"
----
================================================================================================================================
====

:on error exit -- Instructs SQLCMD to abort execution as soon as an erroneous batch is encountered

GO
```

SqlCmd is a SQL Server utility with a unique scripting syntax that provides us with two key advantages over deploying with plain old T-SQL:

- The `:on error exit` directive ensures that, if any statement within the script raises an unhandled exception, execution is immediately halted at the current batch (rather than simply continuing to the next batch after GO)
- Support for variables which can be passed via the SQLCMD.EXE command line tool. Those variables can be used throughout your scripts using the `$(VariableName)` syntax. This is a feature that we've exploited extensively within ReadyRoll's Octopus Deploy integration.

XACT_ABORT to control execution flow

The next thing you'll notice is the use of the `SET XACT_ABORT ON` predicate. According to MSDN:

> *When SET XACT_ABORT is ON, if a Transact-SQL statement raises a run-time error, the entire transaction is terminated and rolled back. When SET XACT_ABORT is OFF, in some cases only the Transact-SQL statement that raised the error is rolled back and the transaction continues processing.*

To avoid the need to add excessive amounts of error-handling logic to your scripts, **we explicitly set this predicate ON as part of the build** and prevent you from setting it to OFF within migrations and programmable objects/additional scripts (unless the *Custom* transaction-handling mode is used; see below *Disabling Automatic Transaction Handling*).

Pre & Post-Deployment script XACT_ABORT behavior

In spite of there being no automatic transaction handling for the files contained within the *Pre-Deployment* and *Post-Deployment* folders, ReadyRoll also sets XACT_ABORT to the ON setting within these scripts for consistency of deployment behavior. However, as some features of the T-SQL language (such as the *sp_fulltext_load_thesaurus_file* statement) require this setting to be OFF, it may be necessary to include a SET XACT_ABORT OFF statement in the header of your Pre/Post Deployment scripts. Note that, for these types of scripts, it is not necessary to include the *Custom* header metadata described further below.

Putting it together

The combination of SqlCmd and XACT_ABORT ensures that, if an exception occurs at any point during deployment, the execution is halted and the connection dropped at the current statement.

Additionally, scripts that have been executed thus far will be rolled-back:

Transaction handling sample deployment script

```
:setvar DatabaseName "AdventureWorks"

:on error exit -- Instructs SQLCMD to abort execution as soon as an erroneous batch is encountered

SET XACT_ABORT ON;
BEGIN TRANSACTION;

------------------ BEGIN MIGRATION: "0016_20131129-1641_User.sql" --------------------
CREATE TABLE t1
    (a INT NOT NULL PRIMARY KEY);

INSERT INTO t1 VALUES (1);
INSERT INTO t1 VALUES (2);
GO
------------------- END MIGRATION: "0016_20131129-1641_User.sql" ---------------------
INSERT [$(DatabaseName)].[dbo].[__MigrationLog] ([migration_id], [script_checksum], [script_filename],
[complete_dt], [applied_by], [deployed])
VALUES (CAST ('d9a91eef-2db8-4911-868b-b29d3d86bae0' AS UNIQUEIDENTIFIER),
'82E9FEA4D0637C8F09DC4D67F4EEC0ED2F706AED8ADFDFA845FFE06188EDE75A', '0016_20131129-1641_User.sql',
SYSDATETIME(), SYSTEM_USER, 1);
GO

------------------------ BEGIN PROGRAMMABLE OBJECT: "dbo.v1.sql" ----------------------------
IF OBJECT_ID('[dbo].[v1]') IS NULL
  EXEC('CREATE VIEW [dbo].[v1] AS SELECT 1 AS Dummy'); -- PLACEHOLDER ONLY - Specify object parameters/body
within the ALTER statement
GO

ALTER VIEW [dbo].[v1]
  AS SELECT a, b FROM [t1] -- Will fail because `[b]` is an invalid column
GO
------------------------- END PROGRAMMABLE OBJECT: "dbo.v1.sql" -----------------------------
INSERT  [$(DatabaseName)].[dbo].[__MigrationLog] ([migration_id], [script_checksum], [script_filename],
[complete_dt], [applied_by], [deployed])
VALUES  (CAST ('0fa6126b-1e3b-438e-8915-72954309cce6' AS UNIQUEIDENTIFIER),
'B352156857D4BCA92A1DD2931642CECC43FBEB86C5BE305339A28F79EE9002FD', 'dbo.v1.sql', SYSDATETIME(), SYSTEM_USER,
1); -- This line will not be reached due to the error in "dbo.v1.sql"
GO

COMMIT TRANSACTION; -- This line will not be reached due to the error in "dbo.v1.sql"
```

One thing conspicuously missing here is a `ROLLBACK` statement at any point. This is because a rollback is implicit in the use of the XACT_ABORT predicate: when an unhandled error occurs, SQL Server immediately performs a rollback and raises an error to the client (in our case, `SQLCMD.EXE`).

Controlling execution flow with TRY/CATCH blocks

For the most part, we hope that the structure we've added around your migrations means you'll never need to worry about how transactions are handled. However there are some use cases for which you might want to exercise greater control over the flow of execution.

For example, say you want to write some details of an error to a log table. Typically, your log records would simply be rolled-back along with your other changes when an error occurs.

But with a TRY/CATCH block, you have the ability to capture exceptions that are raised within your batch and issue a `ROLLBACK` yourself. For example:

Try/Catch sample deployment script

```
:setvar DatabaseName "AdventureWorks"

:on error exit

SET XACT_ABORT ON;
BEGIN TRANSACTION;

------------------ BEGIN MIGRATION: "0016_20131129-1641_User.sql" --------------------
CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);

BEGIN TRY
  INSERT INTO t1 VALUES (1);
  INSERT INTO t1 VALUES (1); -- Will fail because ID=1 already exists in the table
END TRY
BEGIN CATCH
  IF @@TRANCOUNT > 0
      ROLLBACK TRANSACTION; -- We must roll-back because our transaction is doomed at this point

  BEGIN TRANSACTION
      IF OBJECT_ID('error_log') IS NULL
          CREATE TABLE error_log (id INT IDENTITY(1, 1) PRIMARY KEY, msg NVARCHAR(MAX));

      INSERT INTO error_log (msg) VALUES (ERROR_MESSAGE());
  COMMIT TRANSACTION
  RAISERROR(N'An error occurred. Halting deployment.', 16, 127, N'UNKNOWN') WITH NOWAIT;
END CATCH
------------------- END MIGRATION: "0016_20131129-1641_User.sql" ---------------------
GO
INSERT  [$(DatabaseName)].[dbo].[__MigrationLog] ([migration_id], [script_checksum], [script_filename],
[complete_dt], [applied_by], [deployed])
VALUES  (CAST ('d9a91eef-2db8-4911-868b-b29d3d86bae0' AS UNIQUEIDENTIFIER),
'06D29866BEE2749E96C842DDE120CBBA96624D4B2882AFFD97AA8298994AD9A9', '0016_20131129-1641_User.sql',
SYSDATETIME(), SYSTEM_USER, 1); -- This line will not be reached due to the error in the above migration
GO

COMMIT TRANSACTION; -- This line will not be reached due to the error in the above migration
```

This code will raise an exception at line 13, causing the prior operations in the batch to be rolled-back. However the new row in the `error_log` will persist even after the script is halted (via the `RAISERROR`), because it occurs in a separate transaction to the statements within the `TRY` block.

> **Note that not all exceptions can be contained by TRY/CATCH blocks**: if for some reason your batch fails to compile, the error will not be caught and the exception will instead be thrown to the client. For example, a primary key violation error (like you might get with an `INSERT` state ment) *will* be caught by the `CATCH` block, however a reference to a non-existent object will not.

How are syntax errors handled?

ReadyRoll runs all of your migrations through the T-SQL compiler during project build, so provided your statements are not being executed as dynamic SQL (eg. using EXEC / `sp_execute`), you should not receive syntax errors at deployment time.

However testing your database project by doing a full deploy to a test server is still the most foolproof way of validating your T-SQL migrations.

Disabling Automatic Transaction Handling

If your use case dictates that the migration be executed *outside* of a user transaction, or in a transaction that is isolated from any others in the deployment, you can control this behavior at the script level by switching to `Custom` transaction handling:

```
-- <Migration ID="(migration id)" TransactionHandling="Custom" />
```

To disable transaction handling within a programmable object or additional script file, the Migration metadata is also used, but with the ID attribute omitted from the element:

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

You might want to use `Custom` mode if:

- You have a long-running operation like a `BULK INSERT` and you want to commit batches of rows at a time
- You want to automate the deployment of a server-level object such as a linked server
- You need to perform an `ALTER DATABASE` operation on the current database
- You need to add full-text indexes to your database, or perform some other operation which cannot be done within a user transaction

Prior to executing your Custom-flagged migration, ReadyRoll will `COMMIT` any open transactions. After executing that migration, a new transaction will be opened for any remaining migrations that are pending deployment.