PowerShell SQL Deployment – Part 2: Azure DevOps

By Andrew Hinkle

[*DogFoodCon*](https://dogfoodcon.com/) *2019 is October 3rd – 4th.* [*Ben Miller*](https://dogfoodcon.com/speakers/#sz-speaker-9d0a0d9d-bef7-40b6-a81a-ff09cdefa74e) *will be presenting* [*[DevOps] DBAs working Smarter not Harder with PowerShell*](https://dogfoodcon.com/sessions/#sz-session-126790) *and* [*[SQL/BI] Inside the DBAtools PowerShell Module*](https://dogfoodcon.com/sessions/#sz-session-126789) *.*

This article is a follow-up to Part 1 where you learned how to create a couple PowerShell scripts to call the Invoke-Sqlcmd to apply SQL files to a server instance's database. That's nice, but to get the greatest reward of this feature we're going to implement the [PowerShell script](https://docs.microsoft.com/en-us/azure/devops/pipelines/scripts/powershell?view=azure-devops) within your [Azure DevOps Build/Release pipeline](https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops). Once you learn how to setup these Build and Release definitions in the prototype you'll be one step closer to a push button deployment and eventually an automated [Continuous Deployment](https://en.wikipedia.org/wiki/Continuous_deployment) process. This is an advanced topic for those not familiar with the Azure DevOps Build/Release pipeline.

# Assumptions

1. Source Control
   1. TFVC (Team Foundation Version Control) is demoed in this article
   2. If you use [GitHub](https://docs.microsoft.com/en-us/azure/devops/pipelines/create-first-pipeline?view=azure-devops&tabs=tfs-2018-2) or another source control source allowed in the pipeline adjust the steps to meet your needs
2. [Azure DevOps Build Pipelines](https://www.azuredevopslabs.com/labs/devopsserver/build/) has been setup
3. Azure Agent Pools has been setup
4. You know which agents are in the Agent Pools and most importantly which service accounts are running the agents
   1. Check out the [Service Accounts Security Discussion](#_Service_Accounts_Security) below for critical information regarding security
5. Basic understanding of the Azure DevOps Build/Release pipeline
6. Know how to create a Build and Release definition

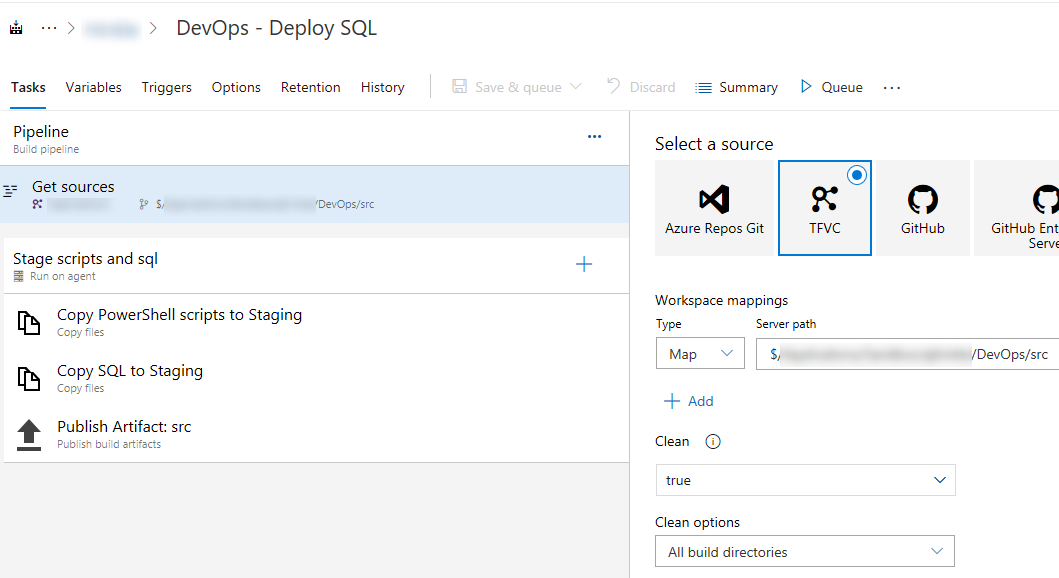
# Check-in PowerShell scripts into source control (TFVC)

In your source control under the equivalent of a Sandbox folder create the following folder structure. These are the same files we created in Part 1. This folder structure is simple for this prototype. You'll need to work with your team on where the scripts folder should be stored since they won't change much if ever. For the sql folder you may want to standardize on a release folder with a folder for each release number containing the sql files (sql\R19.12\DATABASE1\\*). When you have variables changing between releases you add [queue time variables](https://docs.microsoft.com/en-us/azure/devops/pipelines/process/variables?view=azure-devops&tabs=yaml%2Cbatch#allow-at-queue-time) and reference them in the build definition so the build definition doesn't have to change.

1. Create folder structure in source control
   1. DevOps\src\scripts\COMPANYNAME.DevOps.Sql.psm1
   2. DevOps\src\scripts\DeploySql.ps1
   3. DevOps\src\sql\DATABASE1\00.Setup.sql
   4. DevOps\src\sql\DATABASE1\01.Deploy.sql
   5. DevOps\src\sql\DATABASE1\99.TearDown.sql
   6. DevOps\src\sql\DATABASE2\00.Setup.sql
   7. DevOps\src\sql\DATABASE2\01.Deploy.sql
   8. DevOps\src\sql\DATABASE2\99.TearDown.sql
2. Check-in code

# Create the Build Definition

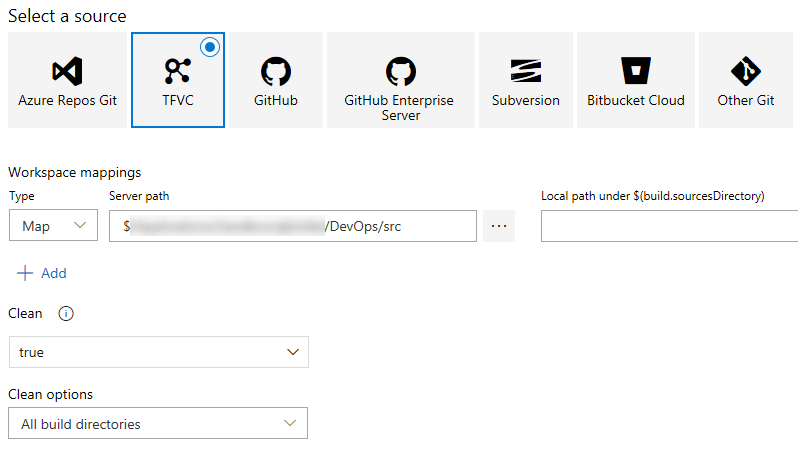
We'll create an Azure DevOps Build Definition that will get the PowerShell scripts and SQL files from source control, copy them to a folder, and then publish them so the files are available for the Release Definition to deploy them by environment. Eventually when you are comfortable with the process you can setup the Build Definition to automatically create when code is checked into the source control location satisfying Continuous Integration. For now we'll do it manually. Create a Build Definition following the template below.



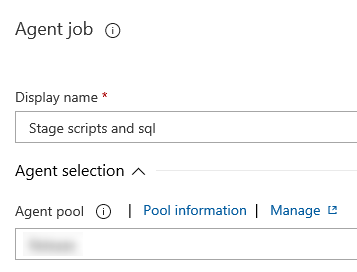
Title: DevOps - Deploy SQL

Tasks

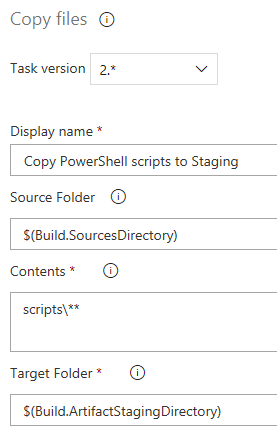
1. Pipeline
   1. Get sources
      1. Select your source: TFVC (Team Foundation Version Control)
   2. Workspace mappings
      1. Type: Map
      2. Server path: {select the path where you checked in the "src" folder}
      3. Clean: true



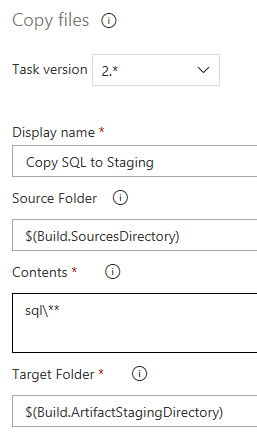
1. Stage scripts and sql (Run on agent)
   1. Display name: Stage scripts and sql
   2. Agent pool: {name of agent pool}
      1. See the [Service Accounts Security Discussion](#_Service_Accounts_Security) below



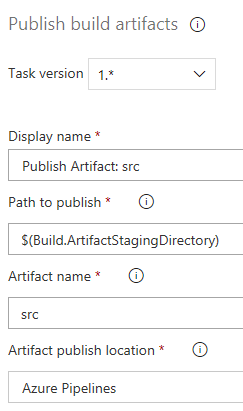
1. Copy PowerShell scripts to Staging
   1. Source Folder: $(Build.SourcesDirectory)
   2. Contents: scripts\\*\*
   3. Target Folder: $(Build.ArtifactStagingDirectory)



1. Copy SQL to Staging
   1. Source Folder: $(Build.SourcesDirectory)
   2. Contents: sql\\*\*
   3. Target Folder: $(Build.ArtifactStagingDirectory)

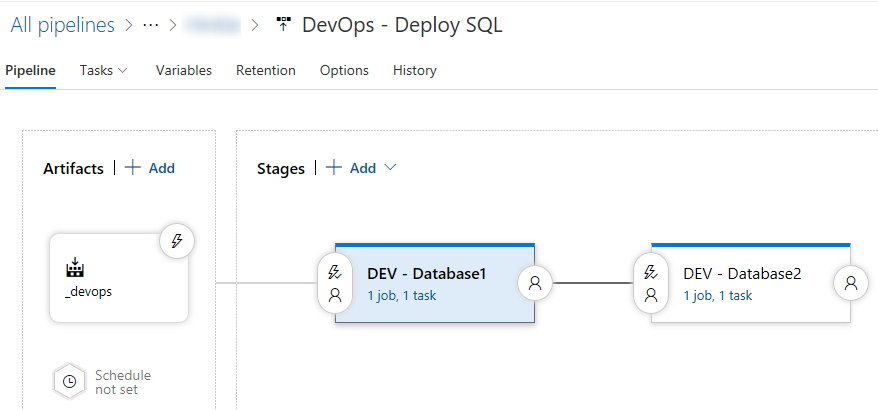


1. Publish Artifact: src
   1. Path to publish: $(Build.ArtifactStagingDirectory)
   2. Artifact name: src
   3. Artifact publish location: Azure Pipelines



# Create the Release Definition

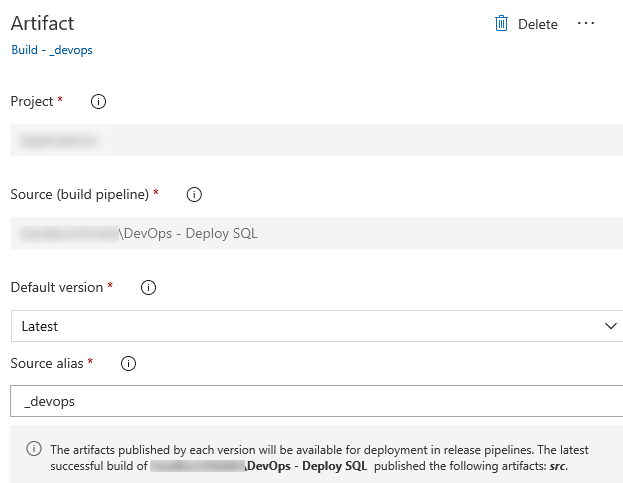
We'll create an Azure DevOps Release Definition that picks up the staged artifacts which are the PowerShell scripts and SQL files and deploy them to the DEV environments. In this scenario the SQL Server instance for Database1 and Database 2 are on different DEV servers. We'll need to use variables scoped to each environment. Also, let's deploy the databases in order and stop the deployment if the first database fails to deploy. Once the definition is setup and you are confident in the process you could automate the creation of the Release Definition whenever the associated Build finishes satisfying a Continuous Deployment strategy. For now we'll leave the process manual. Create a Release Definition following the template below.



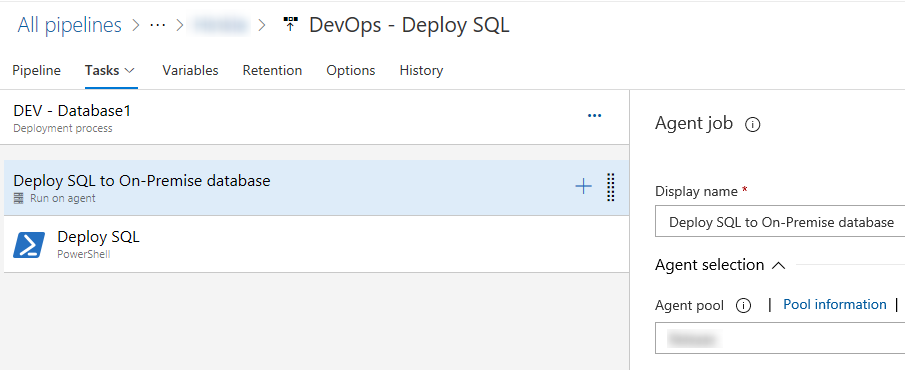
Title: DevOps - Deploy SQL

Pipeline

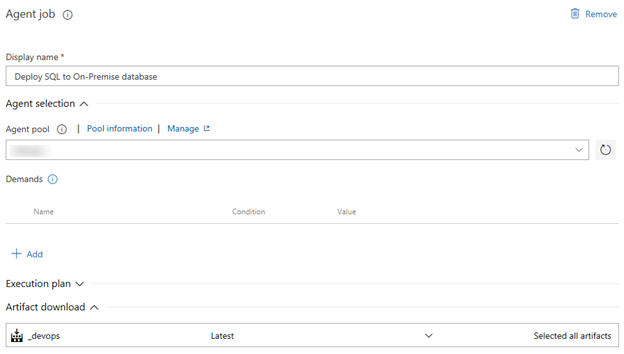
1. Artifacts
   1. Project: {Select the same Project the Build Definition was created under}
   2. Source: {Select the Build Definition you created}
   3. Default Version: Latest
   4. Source Alias: \_devops
      1. Changing the Alias allows you to clone the Release Definition for feature environments without having an Alias that was named as the original definition



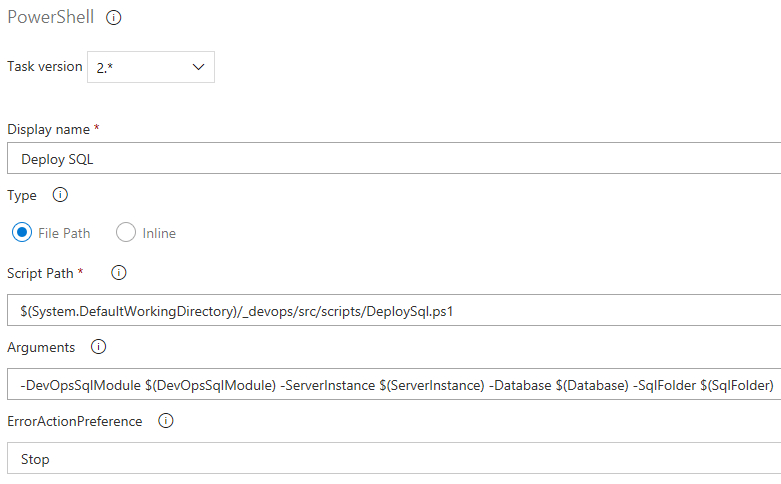
1. Tasks - DEV - Database1



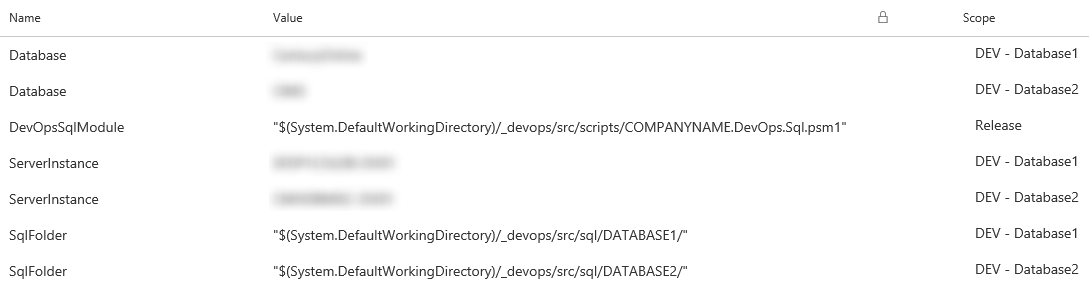
* 1. Agent job
     1. Display name: Deploy SQL to On-Premise database
     2. Agent pool: {name of agent pool}
        1. See the [Service Accounts Security Discussion](#_Service_Accounts_Security) below
     3. Artifact download: \_devops



* 1. PowerShell  
     If you are deploying to an Azure database then choose Azure PowerShell, the Invoke-SqlCmd works the same. You still need to have the Service Account and permissions setup.
     1. Display name: Deploy SQL
     2. Type: File Path
     3. Script Path:
        1. $(System.DefaultWorkingDirectory)/\_devops/src/scripts/DeploySql.ps1
     4. Arguments:
        1. -DevOpsSqlModule $(DevOpsSqlModule) -ServerInstance $(ServerInstance) -Database $(Database) -SqlFolder $(SqlFolder)
           1. The $(\*) are environment variables setup in the Variables tab that you'll setup shortly
     5. ErrorActionPreference: Stop
        1. Do not continue if the PowerShell exits with an error



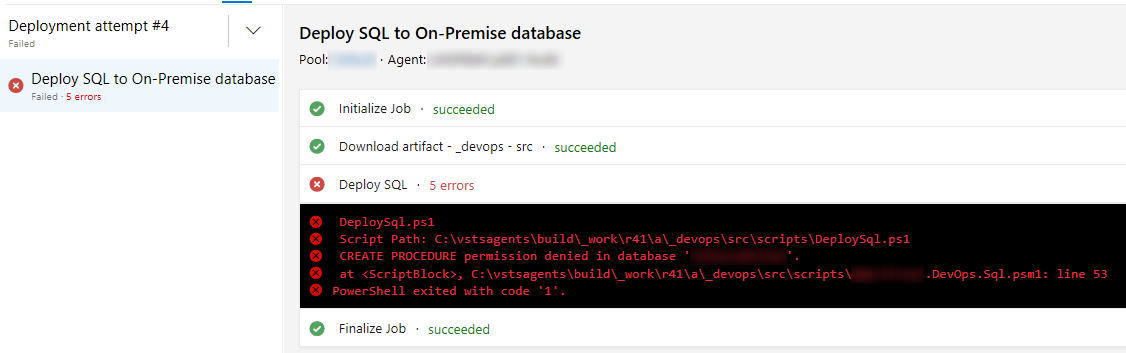
1. Tasks - DEV - Database2
   1. Repeat the steps for "Tasks - DEV - Database1" updating the title of the task
2. Variables (Pipeline Variables)  
   The scope specifies which task the variables is applied to. Release means all tasks. The module is the same for both tasks so we set the scope to Release while the rest of the variables are scoped to the environment.
   1. Database
      1. Value: {Database1}
      2. Scope: DEV - Database1
   2. Database
      1. Value: {Database2}
      2. Scope: DEV - Database2
   3. DevOpsSqlModule
      1. Value: "$(System.DefaultWorkingDirectory)/\_devops/src/scripts/COMPANYNAME.DevOps.Sql.psm1"
      2. Scope: Release
   4. ServerInstance
      1. Value: {ServerInstance1}
      2. Scope: DEV - Database1
   5. ServerInstance
      1. Value: {ServerInstance2}
      2. Scope: DEV - Database2
   6. SqlFolder
      1. Value: "$(System.DefaultWorkingDirectory)/\_devops/src/sql/DATABASE1/"
      2. Scope: DEV - Database1
   7. SqlFolder
      1. Value: "$(System.DefaultWorkingDirectory)/\_devops/src/sql/DATABASE2/"
      2. Scope: DEV - Database2



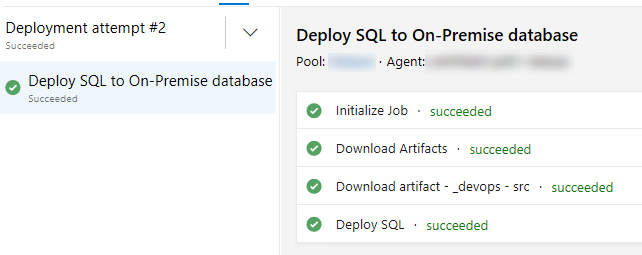
1. Save and Queue

# Error Results

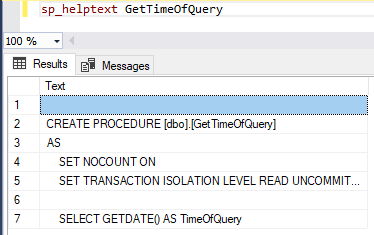
From the Part 1 article remember that we caught exceptions thrown when trying to apply a SQL file. We built the error message using the special Azure DevOps codes and returned the stack trace. By exiting with code "1" from the PowerShell script the task was triggered as an error and stopped processing. In this case we did not have enough permissions to create the test stored procedure.



# Success Results

When the pipeline finally worked it felt good to see all of those successful check marks!

**DON'T ASSUME IT WORKED!!!** Just because the release succeeded double-check by running the "sp\_helptext GetTimeOfQuery", that's the name of the stored procedure we created in the Part 1 article. Cool, all looks good.



# Service Accounts Security Discussion

Critically important! The (OnPrem or Azure) Active Directory user account associated with the agents running in the Agent pool must have permission to apply the SQL scripts. These PowerShell scripts are setup with the assumption that you are using [Integrated Security](https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/authentication-in-sql-server) to deploy the SQL files. This saves you a ton of effort with encryption by abstracting it out to the Active Directory. Usually it's easier to start with giving the account db\_owner permissions on the database you're deploying to just to prove out the concept.

Work with your DBAs and Ops to create Service Accounts that are strictly used to deploy the SQL files. Limit the service accounts by environment (DEV/QA/PROD as an example). DBAs should restrict the permissions of the service accounts to just what is necessary for the deployments.

The Invoke-Sqlcmd allows for other methods such as connection strings with the user account and password, but I prefer Integrated Security to avoid leaking passwords and other secrets. Research and understand the security risks with the scripts and how they connect.

# Conclusion

I have demonstrated how to create the Azure DevOps Build/Release pipeline implementing PowerShell to deploy SQL to an OnPrem or Azure database. Next steps include implementing this process in a Sandbox to learn the process yourself. Work with the DBAs and Ops to create Service Accounts to tighten permissions and security.

To learn more about how to create your SQL files automatically by shredding the results published from a DACPAC created from a Database Project attend: [*DogFoodCon*](https://dogfoodcon.com/) *2019 is October 3rd – 4th.* [*Ben Miller*](https://dogfoodcon.com/speakers/#sz-speaker-9d0a0d9d-bef7-40b6-a81a-ff09cdefa74e) *will be presenting* [*[DevOps] DBAs working Smarter not Harder with PowerShell*](https://dogfoodcon.com/sessions/#sz-session-126790) *and* [*[SQL/BI] Inside the DBAtools PowerShell Module*](https://dogfoodcon.com/sessions/#sz-session-126789) *.*

Did you know how to create a Build/Release pipeline to deploy SQL? Do you prefer "[Deploying Databases using Azure DevOps Pipelines](https://www.danylkoweb.com/Blog/deploying-databases-using-azure-devops-pipelines-NQ)" following the steps outlined by Jonathan Danylko? Sure is nice to have options! If you weren't familiar with the DevOps pipeline before are you interested in learning more now?