**NetCompany**

**Arm templates – Tips & tracks**

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

|  |  |
| --- | --- |
| **Version:** | 0.1 |
| **Status:** | Draft |
| **Approver:** | [Approver name] |
| **Author:** | Kenneth Rasmussen |



**Document history**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version | Date | Author | Status | Comments |
| 0.1 | 25-10-2019 |  | Draft |  |

**References**

|  |  |  |  |
| --- | --- | --- | --- |
| Reference | Title | Author | Version |
|  |  |  |  |

Table of contents

[1 No help 3](#_Toc256000000)

# Introduction

In the following I will go through different things I learned from trying to establish good deployment procedures for Azure Resources.

# Nested Templates

When deploying many resources, ARM templates can quickly become very large, hard to handle for visual studio and it can be difficult to keep an overview of ones resources. This is especially true, when working with Logic Apps, since Logic App templates tend to get long and complicated. The code language for Logic Apps is actually very similar to the language of Arm Templates with some subtle differences, and it is almost like arm templates inside arm templates.

To help keep things structured, but still deploy several Logic Apps or other resources from the same template, I have found it very helpful to use nested templates – the ARM resource is of type “Microsoft.Resources/deployments”, and it is really a sub-deployment, which is called from inside a Master ARM template.

## Staging and templateLink

The problematic part of using nested templates is, that the nested template needs to be referenced with a uri (templateLink), so the deployment files need to be staged somewhere. The way, we have been dealing with this, is by temporarily copying files to a container in Blob Storage on Azure and reference using the file name and SasKey for the blob storage.

When deploying from an ARM deployment project in visual studio, you can add the two parameters **“\_artifactsLocation”** and **“\_artifactsLocationSasToken”**, and when these are present visual studio stage the files in a blob storage of your choosing, and uses these variables to store the location. In DevOps we stage files to blob storage by using Azure File Copy as the first step of the deployment, and here is a similar way to populate the location and token parameters, which I will get back to later. I found that DevOps staging adds a “/” in the end of the location uri, whereas Visual Studio doesn’t, so to make it possible to use the same template for both, I have added the parameter “\_devOpsDeployment”, which should be set to true in DevOps and false in Visual Studio, then instead of using “\_artifactsLocation” directly, we construct and use the variable

"templateLocation": "[if(parameters('\_devOpsDeployment'), parameters('\_artifactsLocation'),

concat(parameters('\_artifactsLocation'), '/'))]"

This seems unnecessary, but this is a working solution, and the best I have come up with. Now using this we can reference our nested deployment templates like this:

"templateLink": {

          "uri": "[concat(variables('templateLocation'), 'LogicApps/ExampleLogicApp.json' ,

parameters('\_artifactsLocationSasToken'))]"

        },

## Passing parameters

When using nested template, you can either add a separate parameter file for the nested template using **“parameterLink”**, or as we have been doing, have one parameter files (or parameters from DevOps) and pass these parameters on to the nested template. To do this, you simple reference each parameter in the nested template, and define which value from your master template, you want to pass to it

"parameters": {

  "ArmTemplateParameter": {

    "value": "[parameters('ArmTemplateParameter')]"

  }

}

## Using parameters in Logic Apps

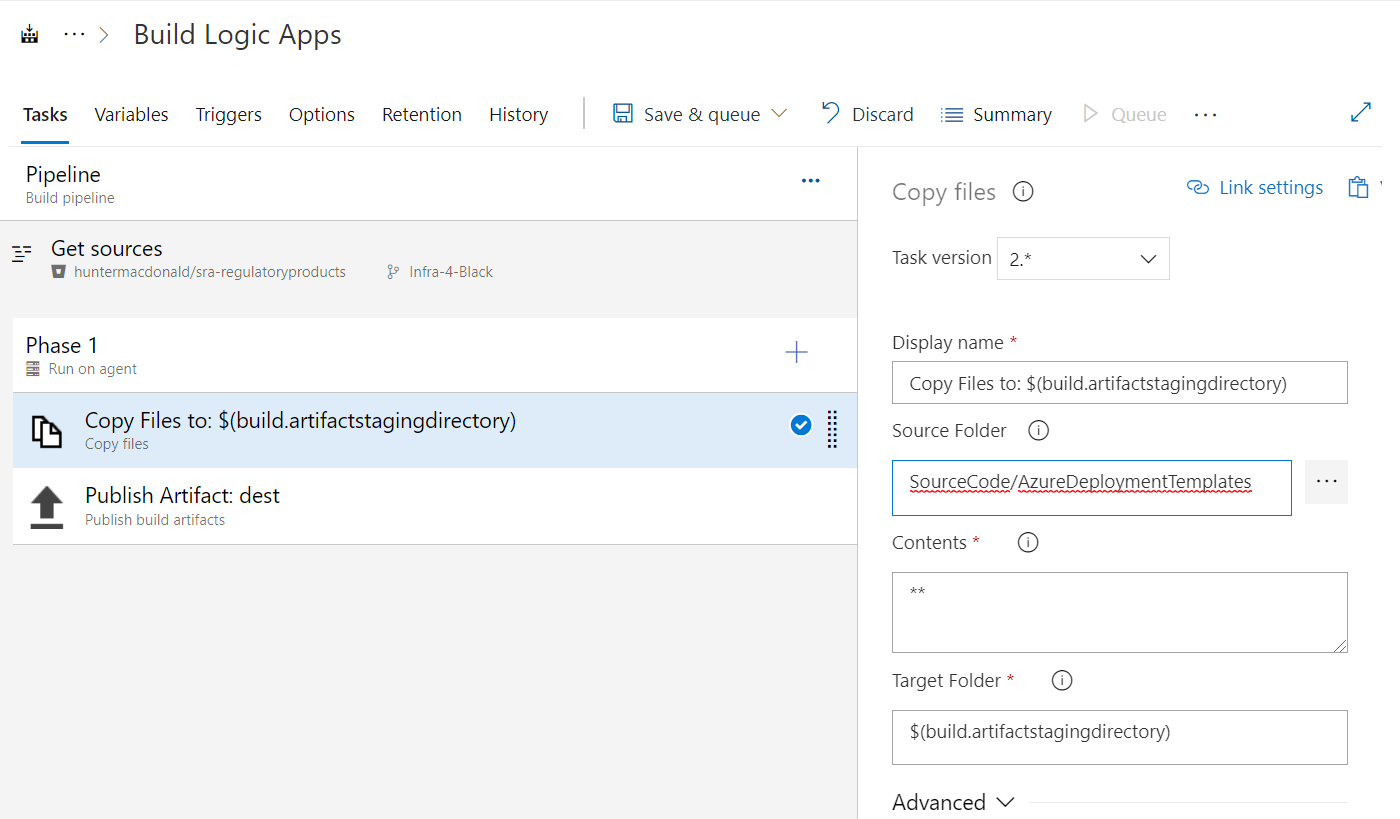
When developing and testing Logic Apps, it is easiest to use the visual Logic App editor on the Azure Portal or through visual Studio, and when you are satisfied with the Logic App, you simply switch to code view and copy the Logic App definition into the example logic app. Now you only need to parametrize The Logic App. Inside the logic there is a seperate parameter section, and the syntax for using parameters in the Logic App is different than the Arm Template syntax.

It is bad practice to use the Arm Template syntax inside Logic Apps, and this can also cause unwanted problems, because the normal Arm Template syntax is used for other things inside a Logic App. Therefore the best way to parametrize Logic Apps is to pass the Arm Template Parameter to a Logic App parameter, which can then be used inside the Logic App. The name of the parameter can of cause be the same or different.

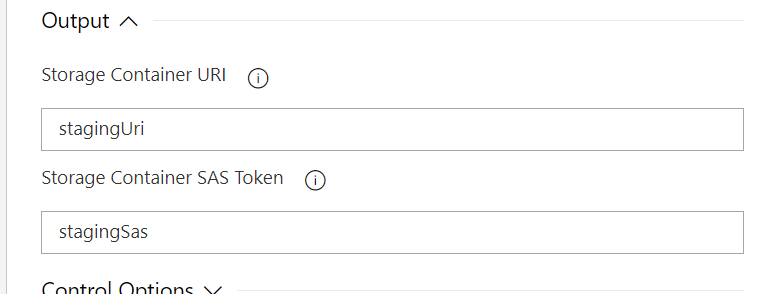
Inside the Logic App, the parameters reference by using the syntax: **@{parameters('LogicAppParameter')}**.

## Staging in DevOps

When deploying through DevOps, the build just consist of copying files from the repository to artifactory and publishing the build artifacts



The release has to tasks, the first part is the Azure File copy job, where you fill out the obligatory fields, but the thing to notice here is, that we need to specify variables names in output section in the button, which will store ArtifactLocation and SasToken in variables with the names you specify.



Lastly the master ARM template is deployed with reference to the build artefact, and here we need to pass the staging parameters on, so in the override template parameters section, we add the following.

-\_artifactsLocation "$(stagingUri)" -\_artifactsLocationSasToken "$(stagingSas)" -\_devOpsDeployment true

With this structure deployment is automated, and each logic app is kept in separate files to give a nice overview. Nested templates and staging in general can be used in for any kind of resource, and for other purposes like keeping API Management policies in separate files etc.

If anyone has any comments or question feel free to reach out to me.