**Azure DevOps pipeline + Terraform Deployment Tutorial**

[Overall Topology](https://www.carbonlogiq.io/post/azure-devops-pipeline-terraform-deployment-tutorial#viewer-0g3qe29073)

[Create the Service Principal](https://www.carbonlogiq.io/post/azure-devops-pipeline-terraform-deployment-tutorial#viewer-vdy9576)

[Create the Terraform State Storage Account](https://www.carbonlogiq.io/post/azure-devops-pipeline-terraform-deployment-tutorial#viewer-ry52k87)

[Set Azure Subscription Permissions](https://www.carbonlogiq.io/post/azure-devops-pipeline-terraform-deployment-tutorial#viewer-f3wrb171924)

[Create the Service Bus Terraform Code](https://www.carbonlogiq.io/post/azure-devops-pipeline-terraform-deployment-tutorial#viewer-ym8sq237839)

[Configure Azure DevOps](https://www.carbonlogiq.io/post/azure-devops-pipeline-terraform-deployment-tutorial#viewer-a1pem355187)

[Run and Test the Pipeline](https://www.carbonlogiq.io/post/azure-devops-pipeline-terraform-deployment-tutorial#viewer-8pmal843376)

[Scaling the Deployment](https://www.carbonlogiq.io/post/azure-devops-pipeline-terraform-deployment-tutorial#viewer-jmsfd1440642)

[Conclusion](https://www.carbonlogiq.io/post/azure-devops-pipeline-terraform-deployment-tutorial#viewer-lavoh79543)

I recently needed to use Terraform to deploy Azure services via Azure DevOps.

Having focused on Bicep for the past couple of years, it’s been a while since I’ve used Terraform, so I was looking for a quick example Azure DevOps (ADO) deployment pipeline.

There are some good references out there, but I felt most of them skimmed over some key configuration, such as Azure subscription permissions or ADO secrets.

This guide will cover everything required to deploy an example Azure Service Bus instance via Terraform and ADO.

In addition to creating an example pipeline, we’ll also add enhanced capabilities, including -

* Hosting the Terraform backend state on Azure blob storage
* Creating a deployment Service Principal + setting RBAC permissions on the Azure subscription
* Create a multi-stage ADO pipeline with an approval step
* Demonstrating how we can scale with multiple deployments

If you’d like to skip straight to the download of the example pipeline and Terraform code, it can be found on [GitHub](https://github.com/andrew-kelleher/terraformadopipeline).

**Overall Topology**

In this diagram, we can see all of the different supporting components required to deploy the Terraform code and the actual Service Bus instance itself -

A screenshot of a computer

AI-generated content may be incorrect.

There are a couple of things worth highlighting -

* **ADO build agent** - more advanced deployment scenarios sometimes require the use of self-hosted agents due to network or security requirements. But for simplicity, we’ll use the ADO-hosted Linux (Ubuntu) build agent, which already has Terraform installed.
* **Git repo** - all of the pipeline config and Terraform code is hosted in a single ADO Git repo
* **Terraform state file** - the state file is hosted in the same Azure subscription, but this could be located on any storage account (permissions permitting)
* **Azure resource group** - the target resource group for the Service Bus instance is *env01-tfdemo-rg*. This will also be created by Terraform.

For easy reference, these are the files we’ll be working with -

A screenshot of a computer

AI-generated content may be incorrect.

The pipeline yaml definitions can reside anywhere within our repo but I prefer to store them in a */deploy*  folder.

Okay, let’s get into it!

**Create the Service Principal**

First things first, a Service Principal (SPN) is required to allow Terraform on the ADO build agent to authenticate against the Azure subscription and create Azure resources -

* Within the Azure Portal, open **Microsoft Entra ID**
* Create a new **App registration**, *tfdemo-spn* (accept the default settings)
* Make a note of the *Application (client) ID* and *Directory (tenant) ID* as we’ll need these later
* Under Certificates & Secrets, create a new secret called *ADO*. Make a note of this value, too.

A screenshot of a computer

AI-generated content may be incorrect.

**Create the Terraform State Storage Account**

Terraform stores its view of our infrastructure and associated metadata in a [state](https://developer.hashicorp.com/terraform/language/state) file. The state file is central to Terraform's operation and must be stored in an appropriate location.

Storing the state file locally is fine for basic testing, but for automated deployments, it must be hosted somewhere the ADO build agent can access. Typically, this is an Azure blob storage account (or an S3 bucket if you’re using AWS).

The steps below describe how to implement this part of the configuration (this is completed manually). Within the Azure Portal -

* Create a new resource group, *tfstate-tfdemo-rg*
* Create a new storage account, *tfstatedemostg*(accept all default configuration settings)
* Create a *tfstate*blob container.

A screenshot of a computer

AI-generated content may be incorrect.

* Under **Access Control (IAM)** for the storage account, grant the *Storage Blob Data Contributor* role to the SPN

A white background with black lines

AI-generated content may be incorrect.

There’s no need to manually create the tfstate file. This will be automatically be created when Terraform first runs within the pipeline.

**Set Azure Subscription Permissions**

In the above step, we granted the SPN permission to write the *tfstate*file to the storage account. The SPN also needs additional permissions to deploy the Azure resources in the wider subscription.

Setting appropriate SPN permissions is a topic in itself, but for now, we’ll grant the SPN *Contributor* permissions on the entire subscription. This allows Terraform on the build agent to create the *env01-tfdemo-rg* resource group and the Service Bus resources inside it -

1. Within the Azure Portal, browse to the target subscription
2. Select **Access control (IAM)**
3. Create a new **role assignment** and assign the SPN the *Contributor* role

A screenshot of a phone

AI-generated content may be incorrect.

An alternate approach is to pre-create the target resource group. This allows the SPN permissions to be more tightly scoped to the existing resource group instead of the entire subscription.

**Create the Service Bus Terraform Code**

In this example, we'll keep it nice and simple and deploy a Service Bus namespace and two queues. This is the Terraform code, *servicebus.tf* -

1. # Service Bus - namespace

2. resource "azurerm\_servicebus\_namespace" "sbus" {

3. name = "${var.project}-${var.environment}-sbns"

4. location = azurerm\_resource\_group.rg.location

5. resource\_group\_name = azurerm\_resource\_group.rg.name

6.

7. tags = local.tags

8.

9. local\_auth\_enabled = false

10. minimum\_tls\_version = "1.2"

11. network\_rule\_set {

12. default\_action = "Allow"

13. public\_network\_access\_enabled = true

14. trusted\_services\_allowed = false

15. }

16. public\_network\_access\_enabled = true

17. sku = "Standard"

18. }

19.

20. # Service Bus - queue01

21. resource "azurerm\_servicebus\_queue" "queue01" {

22. name = "queue01"

23. namespace\_id = azurerm\_servicebus\_namespace.sbus.id

24.

25. default\_message\_ttl = "P14D"

26. dead\_lettering\_on\_message\_expiration = false

27. duplicate\_detection\_history\_time\_window = "PT10M"

28. enable\_batched\_operations = true

29. enable\_partitioning = false

30. lock\_duration = "PT1M"

31. max\_delivery\_count = 10

32. max\_size\_in\_megabytes = 1024

33. requires\_duplicate\_detection = false

34. requires\_session = false

35. }

36.

37. # Service Bus - queue02

38. resource "azurerm\_servicebus\_queue" "queue02" {

39. name = "queue02"

40. namespace\_id = azurerm\_servicebus\_namespace.sbus.id

41.

42. default\_message\_ttl = "P14D"

43. dead\_lettering\_on\_message\_expiration = false

44. duplicate\_detection\_history\_time\_window = "PT10M"

45. enable\_batched\_operations = true

46. enable\_partitioning = false

47. lock\_duration = "PT1M"

48. max\_delivery\_count = 10

49. max\_size\_in\_megabytes = 1024

50. requires\_duplicate\_detection = false

51. requires\_session = false

52. }

53.

Also of interest is the *providers.tf* file. Here, we can see that the state file is configured to be hosted on the storage account we created earlier -

1. terraform {

2. required\_providers {

3. azurerm = {

4. source = "hashicorp/azurerm"

5. version = "~>3.0"

6. }

7. random = {

8. source = "hashicorp/random"

9. version = "~>3.0"

10. }

11. }

12. backend "azurerm" {

13. resource\_group\_name = "tfstate-tfdemo-rg"

14. storage\_account\_name = "tfstatetfdemostg"

15. container\_name = "tfstate"

16. key = "tfdemo.env01.tfstate"

17. }

18. }

19.

20. provider "azurerm" {

21. features {}

22. }

23.

The full Terraform code can be found on [GitHub](https://github.com/andrew-kelleher/terraformadopipeline) and includes the additional files, such as *tfdemo.env01.tfvars* which defines the resource names.

**Configure Azure DevOps**

Within ADO, we need to configure a couple of things to get everything working.

**Create a Variable Group**

The variable group contains the details of the Service Principal we created earlier. Terraform will use these values to authenticate against the target Azure subscription -

* Create a variable group, *Terraform\_SPN* , within **Pipelines → Library**
* Create the following variables using the SPN’s values from earlier
* **ARM\_CLIENT\_ID**
* **ARM\_CLIENT\_SECRET**
* **ARM\_SUBSCRIPTION\_ID**
* **ARM\_TENANT\_ID**

These variable names are of special [significance](https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/guides/service_principal_client_secret#configuring-the-service-principal-in-terraform) to Terraform. When set as environment variables within the ADO build agent, Terraform will automatically attempt to authenticate against Azure using their values.

Your variable group should look something like this -

A screenshot of a computer

AI-generated content may be incorrect.

**Create the ADO Environment**

By creating an ADO [environment](https://techcommunity.microsoft.com/t5/healthcare-and-life-sciences/azure-devops-pipelines-environments-and-variables/ba-p/3707414) and referencing it within our pipeline, we can add an approval step.

When deploying the pipeline, we want an opportunity to review the output from the **Terraform Plan** stage. Only once we’re comfortable with the planned changes do we approve the **Terraform Apply** stage to create the resources.

To create an ADO environment -

* Within **Pipelines → Environments** select **New Environment**
* Set the name as *env01*  and click **Create**
* Once created, select **Approvals and Checks** and click “**+**”

A screenshot of a computer

AI-generated content may be incorrect.

* Select **Next** and add the required approver, i.e. ourselves (fine-tune these settings as required for your organization)

A screenshot of a computer

AI-generated content may be incorrect.

The created environment ->

A screenshot of a computer

AI-generated content may be incorrect.

**Create the Pipeline**

For this example, I’m leveraging an Azure DevOps yaml [template](https://learn.microsoft.com/en-us/azure/devops/pipelines/process/templates?view=azure-devops&pivots=templates-includes). If we wish to later scale out our deployments with multiple environments, etc., it is much easier to create a single reusable Terraform deployment pipeline.

First, the template deployment pipeline definition, *terraform-template.yml* -

1. parameters:

2. - name: rootFolder

3. type: string

4. - name: tfvarsFile

5. type: string

6. - name: adoEnvironment

7. type: string

8.

9. stages:

10. - stage: 'Terraform\_Plan'

11. displayName: 'Terraform Plan'

12. jobs:

13. - job: 'Terraform\_Plan'

14. pool:

15. vmImage: 'ubuntu-latest'

16. steps:

17. - script: |

18. echo "Running Terraform init..."

19. terraform init

20. echo "Running Terraform plan..."

21. terraform plan -var-file ${{ parameters.tfvarsFile }}

22. displayName: 'Terraform plan'

23. workingDirectory: ${{ parameters.rootFolder }}

24. env:

25. ARM\_CLIENT\_SECRET: $(ARM\_CLIENT\_SECRET) # this needs to be explicitly set as it's a sensitive value

26.

27. - stage: 'Terraform\_Apply'

28. displayName: 'Terraform Apply'

29. dependsOn:

30. - 'Terraform\_Plan'

31. condition: succeeded()

32. jobs:

33. - deployment: 'Terraform\_Apply'

34. pool:

35. vmImage: 'ubuntu-latest'

36. environment: ${{ parameters.adoEnvironment }} # using an ADO environment allows us to add a manual approval check

37. strategy:

38. runOnce:

39. deploy:

40. steps:

41. - checkout: self

42. - script: |

43. echo "Running Terraform init..."

44. terraform init

45. echo "Running Terraform apply..."

46. terraform apply -var-file ${{ parameters.tfvarsFile }} -auto-approve

47. displayName: 'Terraform apply'

48. workingDirectory: ${{ parameters.rootFolder }}

49. env:

50. ARM\_CLIENT\_SECRET: $(ARM\_CLIENT\_SECRET)

51.

... and the pipeline definition, *tfdemo-env01-terraform.yml*, for our specific deployment (*tfdemo-env01-rg*) that calls the template above -

1. name: Terraform - deploy Service Bus to env01

2.

3. trigger: none

4.

5. variables:

6. - group: Terraform\_SPN

7. - name: rootFolder

8. value: '/terraform/'

9. - name: tfvarsFile

10. value: 'tfdemo.env01.tfvars'

11. - name: adoEnvironment

12. value: 'env01'

13.

14. stages:

15. - template: templates/terraform-template.yml

16. parameters:

17. rootFolder: $(rootFolder)

18. tfvarsFile: $(tfvarsFile)

19. adoEnvironment: $(adoEnvironment)

20.

To create the pipeline -

* **Pipelines → New pipeline**
* Select **Azure Repos Git**
* Select the repo, i.e. *tfdemo*

A screenshot of a web page

AI-generated content may be incorrect.

* Select **Existing Azure Pipelines YAML** file. *Note - we select the pipeline .yml rather than the template .yml*

A screenshot of a computer

AI-generated content may be incorrect.

* From the **Run** dropdown, select **Save**

A screenshot of a computer

AI-generated content may be incorrect.

* The default pipeline name of *tfdemo*is fine, but this can be renamed to something more meaningful, i.e. *tfdemo-env01-terraform*

Browsing our list of pipelines, our newly created pipeline is now visible -

A screenshot of a computer

AI-generated content may be incorrect.

That’s it! We’ve created a new pipeline, the variable group and ADO environment.

**Run and Test the Pipeline**

Finally, we're ready to run the pipeline and deploy our Service Bus instance for the first time.

After the pipeline has completed the *Terraform\_Plan* stage, we can validate what Terraform expects to deploy. As the Service Bus instance doesn't yet exist, it correctly states that it will be created -

A screenshot of a computer program

AI-generated content may be incorrect.

Now we've satisfied ourselves that the *Terraform\_Plan* stage is correct, click **Review** and **Approve** on the *Terraform\_Apply* stage.

The pipeline takes a few minutes to complete -

A screenshot of a computer

AI-generated content may be incorrect.

...and the resource group and Service Bus instance are created successfully -

A screenshot of a computer

AI-generated content may be incorrect.

On the first run of the pipeline, you will be prompted to grant access to the *Terraform\_SPN* variable group and the *ENV01* environment. Just click **Permit** when prompted.

**Scaling the Deployment**

In this example, we’ve deployed a single resource group with limited resources. But it’s possible quite easily to scale this approach.

For example, imagine the scenario where we must deploy an application hosting environment. Depending on the application, this could include a multitude of Azure services such as an AKS cluster, storage accounts, Application Gateway, VNET, Azure SQL databases, etc.

These services can all be deployed to the same resource group to provide a self-contained hosting environment, i.e. DEV. We can then take this a step further by deploying additional environments, i.e. UAT, PROD, etc., by reusing the same Terraform and pipeline definitions.

The key to making this work is *parameterising everything* within the Terraform and pipeline definitions.

The diagram below shows how, by copying/pasting our existing definitions, and updating a few key values within them, replica environments can easily be deployed.

A screenshot of a computer

AI-generated content may be incorrect.

**Conclusion**

I hope you found this post useful. In this example, we deployed a simple Terraform definition for an Azure Service Bus instance using an ADO pipeline. This Terraform code can easily be extended to deploy additional Azure services as required.

You may also find the following links useful -

* My GitHub [repo](https://github.com/andrew-kelleher/terraformadopipeline) used for this tutorial
* Microsoft: [Terraform on Azure documentation](https://learn.microsoft.com/en-us/azure/developer/terraform/)
* Microsoft: [Create an Azure resource group using Terraform quickstart](https://learn.microsoft.com/en-us/azure/developer/terraform/create-resource-group)
* Hashicorp: [How to authenticate Terraform to Azure using a SPN](https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/guides/service_principal_client_secret)