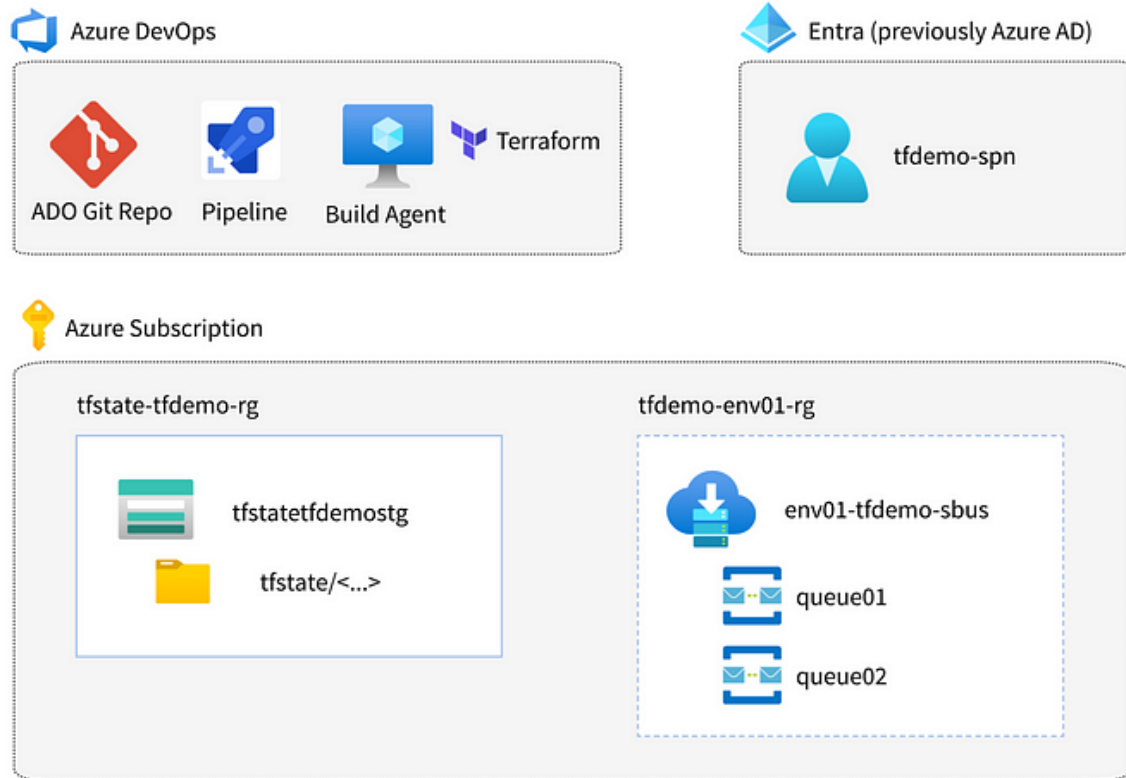


Azure DevOps pipeline + Terraform Deployment Tutorial



In-depth guide to using Azure DevOps to deploy Terraform code to Azure

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.

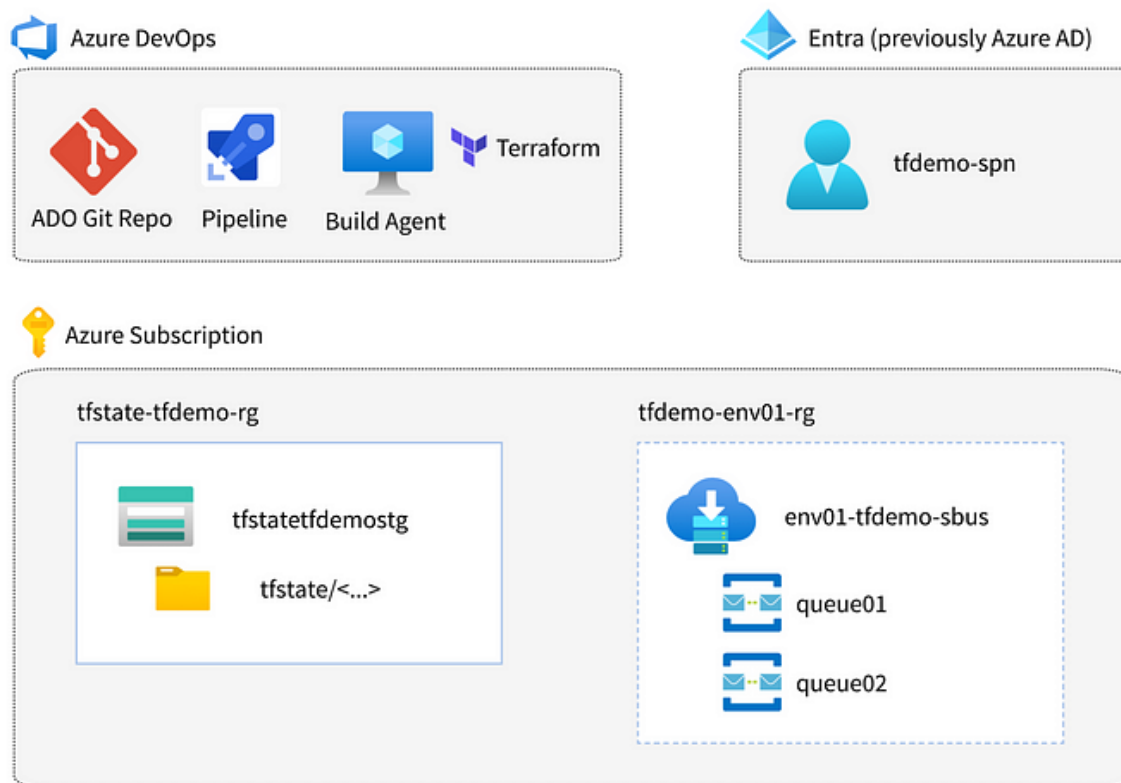
In addition to creating an example pipeline, we'll also add enhanced capabilities, including -

- 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](#).

Overall Topology

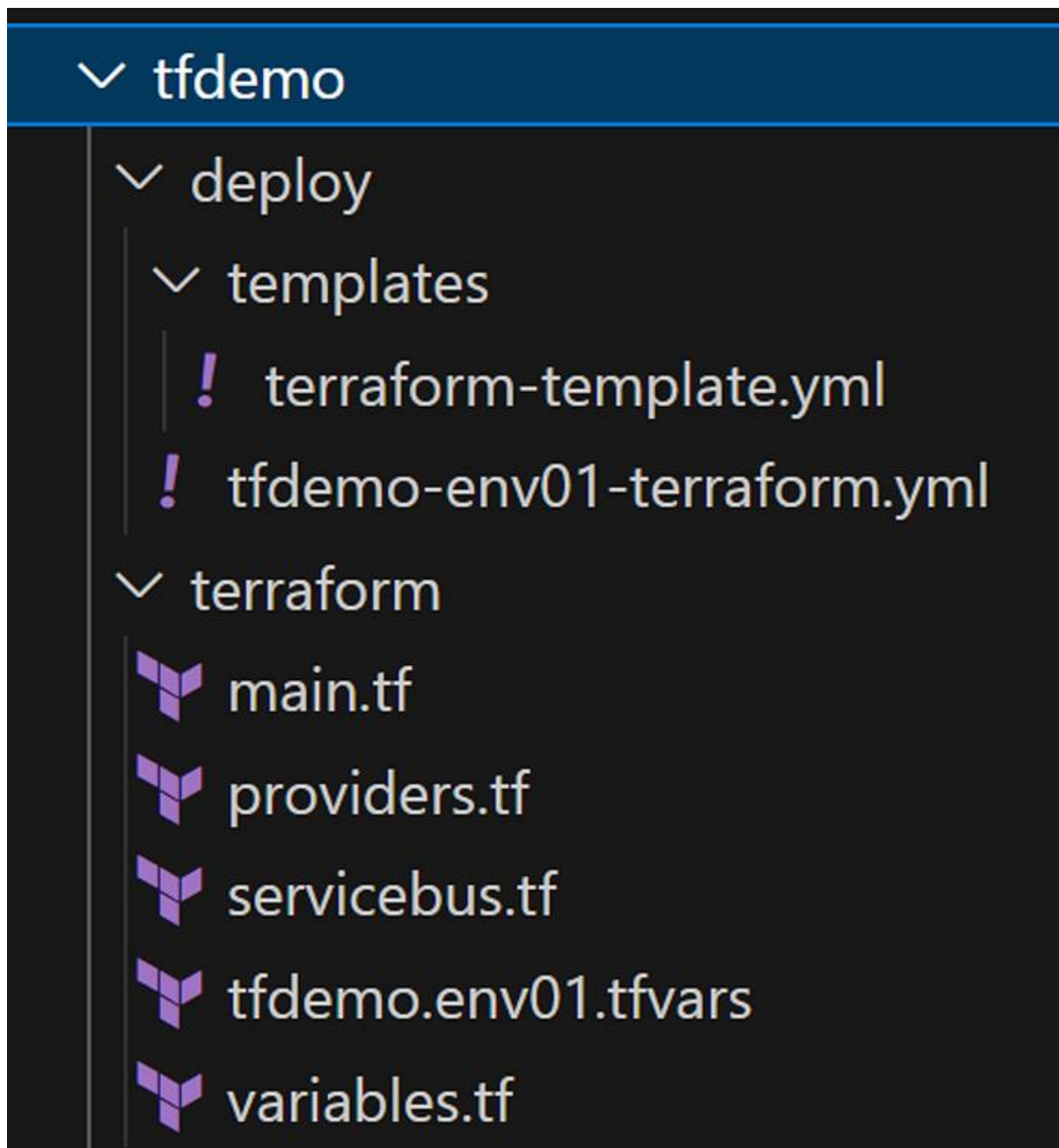
In this diagram, we can see all of the different supporting components required to deploy the Terraform code and the actual Service Bus instance itself -



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 -



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.

Dashboard > Default Directory

Default Directory | App registrations

Microsoft Entra ID

+ New registration Endpoints Troubleshooting Refresh Download ...

All applications Owned applications Deleted applications

tfdemo

1 applications found

Display name	Application (client) ID	Created on	Certificates & secrets
tfdemo-spn	2a9882a4-3583-46a8-a94...	1/8/2024	Current

Create the Terraform State Storage Account

Terraform stores its view of our infrastructure and associated metadata in a [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.

tfstatetfdemostg | Containers

Storage account

Search


+ Container Change access level Restore containers Refresh

Search containers by prefix

Name	Last modified	Anonymous access level
\$logs	1/5/2024, 5:12:35 PM	Private
tfstate	1/5/2024, 5:13:15 PM	Private

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

▼ **Storage Blob Data Contributor (1)**

<input type="checkbox"/>	 tfdemo-spn	App	Storage Blob Data Contributor ⓘ	This resource
--------------------------	--	-----	---------------------------------	---------------

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

▼ **Contributor (1)**

<input type="checkbox"/>	 tfdemo-spn	App	Contributor ⓘ
--------------------------	--	-----	---------------

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* -

Copy# Service Bus - namespace

```
resource "azurerm_servicebus_namespace" "sbus" {
  name          = "${var.project}-${var.environment}-sbns"
  location      = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name

  tags = local.tags

  local_auth_enabled = false
```

```

minimum_tls_version      = "1.2"
network_rule_set {
  default_action          = "Allow"
  public_network_access_enabled = true
  trusted_services_allowed = false
}
public_network_access_enabled = true
sku                           = "Standard"
}

# Service Bus - queue01
resource "azurerm_servicebus_queue" "queue01" {
  name           = "queue01"
  namespace_id = azurerm_servicebus_namespace.sbus.id

  default_message_ttl          = "P14D"
  dead_lettering_on_message_expiration = false
  duplicate_detection_history_time_window = "PT10M"
  enable_batched_operations      = true
  enable_partitioning            = false
  lock_duration                  = "PT1M"
  max_delivery_count             = 10
  max_size_in_megabytes          = 1024
  requires_duplicate_detection   = false
  requires_session               = false
}

# Service Bus - queue02
resource "azurerm_servicebus_queue" "queue02" {
  name           = "queue02"
  namespace_id = azurerm_servicebus_namespace.sbus.id

```

```

default_message_ttl          = "P14D"
dead_lettering_on_message_expiration = false
duplicate_detection_history_time_window = "PT10M"
enable_batched_operations    = true
enable_partitioning          = false
lock_duration                = "PT1M"
max_delivery_count           = 10
max_size_in_megabytes        = 1024
requires_duplicate_detection  = false
requires_session             = false
}

```

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 -

```

Copyterraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "~>3.0"
    }
    random = {
      source = "hashicorp/random"
      version = "~>3.0"
    }
  }
  backend "azurerm" {
    resource_group_name = "tfstate-tfdemo-rg"
    storage_account_name = "tfstatetfdemostg"
    container_name      = "tfstate"
    key                 = "tfdemo.env01.tfstate"
  }
}

```

```
}
```

```
provider "azurerm" {  
  features {}  
}
```

The full Terraform code can be found on [GitHub](#) 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](#) 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 -

Library > Terraform_SPN*

Variable group

Save

Clone

Security

Pipeline permissions

Approvals and checks

Properties

Variable group name

Terraform_SPN

Description

Service Principal credentials for Terraform deployment pipeline

☒ Link secrets from an Azure key vault as variables ⓘ

Variables

Name ↑	Value
ARM_CLIENT_ID	2a9882a4-3583-46a8-a948-7223e29xxxxx
ARM_CLIENT_SECRET	*****
ARM_SUBSCRIPTION_ID	751f15af-6f65-4893-afad-77ab457xxxxx
ARM_TENANT_ID	675eeac2-907e-418b-9996-1262317xxxxx

Create the ADO Environment

By creating an ADO [environment](#) 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 "+"

The screenshot shows the Azure DevOps interface. On the left, the 'env01' environment is selected, with the 'Approvals and checks' tab active. On the right, the 'Add check' dialog is open, showing a search bar for 'Search check types' and a list of check types. The 'Approvals' check type is selected, with the description 'Approvers should grant approval for deployment'.

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

Approvals



Approvers

AA

andrewk andrewk

×

+

Instructions to approvers (optional)

Advanced



☒ Allow approvers to approve their own runs

Control options



Timeout

30

Days

▼

The created environment...

← env01

Test env01 environment for Terraform deployments

Deployments Approvals and checks

Display name

Type

All approvers must approve

Approvals

Create the Pipeline

For this example, I'm leveraging an Azure DevOps yaml [template](#). 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* -

Copyparameters:

- name: rootFolder

type: string

- name: tfvarsFile

type: string

- name: adoEnvironment

type: string

stages:

- stage: 'Terraform_Plan'

displayName: 'Terraform Plan'

jobs:

- job: 'Terraform_Plan'

pool:

vmImage: 'ubuntu-latest'

steps:

- script: |

echo "Running Terraform init..."

terraform init

echo "Running Terraform plan..."

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

displayName: 'Terraform plan'

workingDirectory: \${parameters.rootFolder }

env:

ARM_CLIENT_SECRET: \$(ARM_CLIENT_SECRET) # this needs to be explicitly set as it's a sensitive value

- stage: 'Terraform_Apply'

displayName: 'Terraform Apply'

dependsOn:

- 'Terraform_Plan'

condition: succeeded()

jobs:

- deployment: 'Terraform_Apply'

pool:

vmImage: 'ubuntu-latest'

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

strategy:

runOnce:

deploy:

steps:

- checkout: self

- script: |

echo "Running Terraform init..."

terraform init

echo "Running Terraform apply..."

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

```
displayName: 'Terraform apply'
```

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

```
env:
```

```
ARM_CLIENT_SECRET: $(ARM_CLIENT_SECRET)
```

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

Copyname: Terraform - deploy Service Bus to env01

trigger: none

variables:

- group: Terraform_SPN
- name: rootFolder
value: '/terraform/'
- name: tfvarsFile
value: 'tfdemo.env01.tfvars'
- name: adoEnvironment
value: 'env01'

stages:

- template: templates/terraform-template.yml

parameters:

```
rootFolder: $(rootFolder)
tfvarsFile: $(tfvarsFile)
adoEnvironment: $(adoEnvironment)
```

To create the pipeline -

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

New pipeline

Select a repository

≡ tfdemo



tfdemo

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

Select an existing YAML file



Select an Azure Pipelines YAML file in any branch of the repository.

Branch

 main

▼

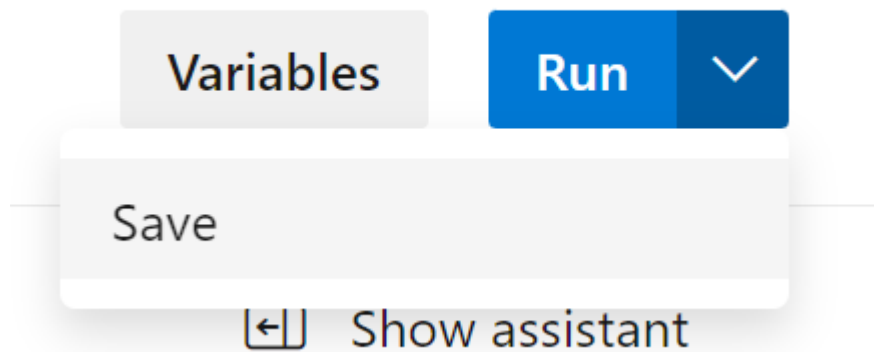
Path

/deploy/tfdemo-env01-terraform.yaml

▼

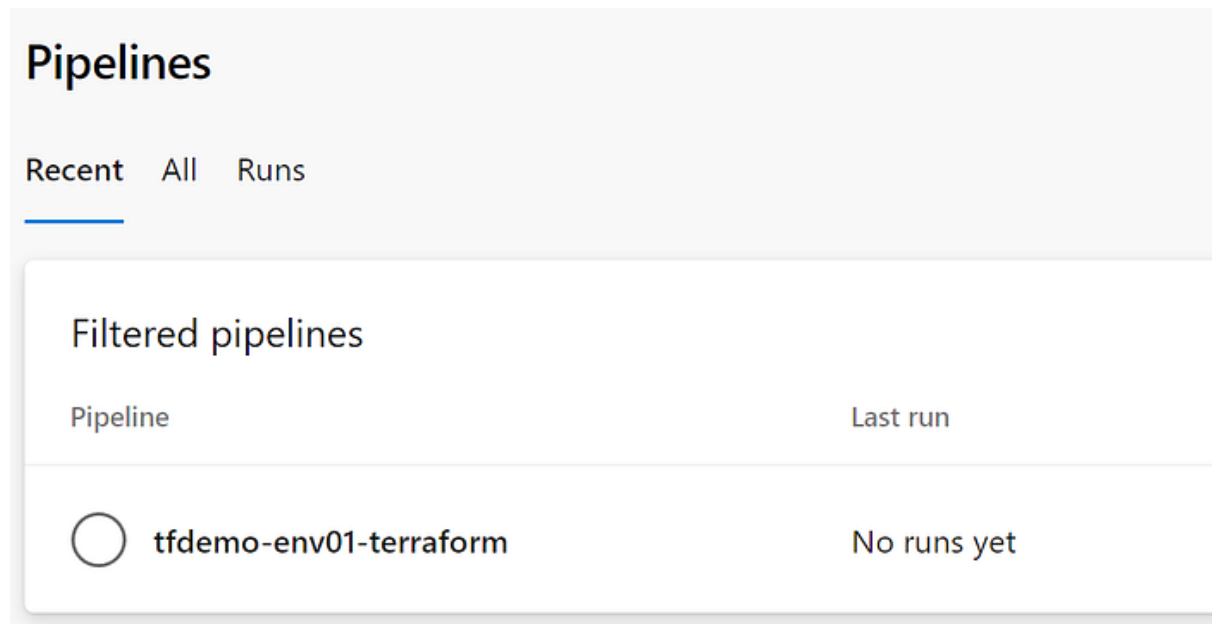
Select a file from the dropdown or type in the path to your file

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



- 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 -



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 -

```

123     + name                                = "queue02"
124     + namespace_id                        = (known after apply)
125     + requires_duplicate_detection        = false
126     + requires_session                    = false
127     + status                              = "Active"
128   }
129
130   Plan: 4 to add, 0 to change, 0 to destroy.
131
132   _____
133
134   Note: You didn't use the -out option to save this plan, so Terraform can't
135   guarantee to take exactly these actions if you run "terraform apply" now.
136
137   Finishing: Terraform plan

```

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 -

← Jobs in run #Terraform ...

tfdemo-env01-terraform

Terraform Plan

✓ Terraform_Plan

1m 50s

Initialize job

<1s

✓ Checkout tfdemo@mai...

1s

✓ Terraform plan

1m 47s

✓ Post-job: Checkout tfd...

<1s

Finalize Job

<1s

Terraform Apply

✓ Terraform_Apply

3m 10s

Initialize job

<1s

✓ Checkout tfdemo@mai...

1s

✓ Terraform apply

3m 8s

✓ Post-job: Checkout tfd...

<1s

Finalize Job

<1s

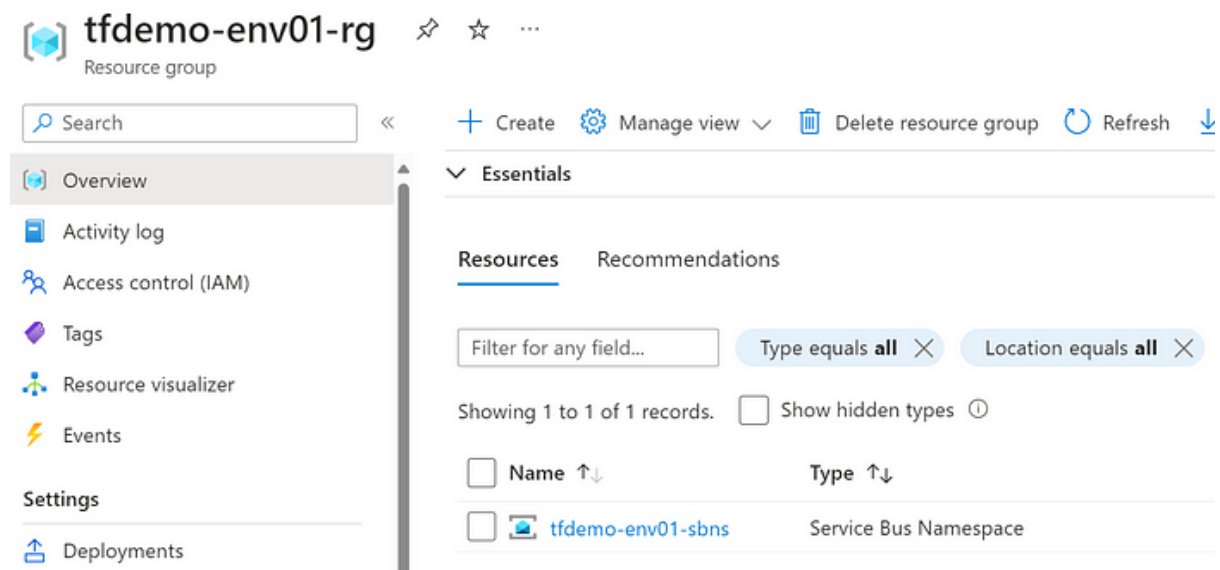
✓ Terraform apply

```

138 azurerm_servicebus_queue.queue01: Still creating... [10s elapsed]
139 azurerm_servicebus_queue.queue02: Still creating... [10s elapsed]
140 azurerm_servicebus_queue.queue01: Still creating... [20s elapsed]
141 azurerm_servicebus_queue.queue02: Still creating... [20s elapsed]
142 azurerm_servicebus_queue.queue01: Still creating... [30s elapsed]
143 azurerm_servicebus_queue.queue02: Still creating... [30s elapsed]
144 azurerm_servicebus_queue.queue02: Still creating... [40s elapsed]
145 azurerm_servicebus_queue.queue01: Still creating... [40s elapsed]
146 azurerm_servicebus_queue.queue02: Still creating... [50s elapsed]
147 azurerm_servicebus_queue.queue01: Still creating... [50s elapsed]
148 azurerm_servicebus_queue.queue01: Still creating... [1m0s elapsed]
149 azurerm_servicebus_queue.queue02: Still creating... [1m0s elapsed]
150 azurerm_servicebus_queue.queue02: Still creating... [1m10s elapsed]
151 azurerm_servicebus_queue.queue01: Still creating... [1m10s elapsed]
152 azurerm_servicebus_queue.queue01: Still creating... [1m20s elapsed]
153 azurerm_servicebus_queue.queue02: Still creating... [1m20s elapsed]
154 azurerm_servicebus_queue.queue02: Still creating... [1m30s elapsed]
155 azurerm_servicebus_queue.queue01: Still creating... [1m30s elapsed]
156 azurerm_servicebus_queue.queue01: Still creating... [1m40s elapsed]
157 azurerm_servicebus_queue.queue02: Still creating... [1m40s elapsed]
158 azurerm_servicebus_queue.queue02: Still creating... [1m50s elapsed]
159 azurerm_servicebus_queue.queue01: Still creating... [1m50s elapsed]
160 azurerm_servicebus_queue.queue02: Still creating... [2m0s elapsed]
161 azurerm_servicebus_queue.queue01: Still creating... [2m0s elapsed]
162 azurerm_servicebus_queue.queue01: Still creating... [2m10s elapsed]
163 azurerm_servicebus_queue.queue02: Still creating... [2m10s elapsed]
164 azurerm_servicebus_queue.queue02: Still creating... [2m20s elapsed]
165 azurerm_servicebus_queue.queue01: Still creating... [2m20s elapsed]
166 azurerm_servicebus_queue.queue02: Creation complete after 2m25s [id=/subscriptions/...
167 azurerm_servicebus_queue.queue01: Still creating... [2m30s elapsed]
168 azurerm_servicebus_queue.queue01: Creation complete after 2m31s [id=/subscriptions/...
169
170 Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
171
172 Finishing: Terraform apply

```


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



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.

