# Module 8 - Infrastructure as Code in Azure DevOps

### 1. Integrating Terraform / ARM / Bicep in Azure Pipelines

Azure DevOps supports **multiple IaC tools** for provisioning and managing Azure resources:

- **Terraform** (HashiCorp, cloud-agnostic)
- **ARM Templates** (Azure-native JSON templates)
- **Bicep** (modern, declarative DSL that compiles to ARM)

You can use these tools inside **YAML pipelines** and **classic pipelines** to:

- Create resource groups, VNets, storage, App Services, AKS, etc.
- Standardize infrastructure definitions
- Enforce repeatable, idempotent deployments

---

#### 1.1 Terraform in Azure Pipelines

Terraform is very commonly used with Azure DevOps.

Typical components:

- terraform init → initialize backend + providers
- terraform plan → show what will change
- terraform apply → apply changes

You usually:

1. Store Terraform configuration (*.tf) in the repo
2. Use Azure Storage as remote backend (for state)
3. Use Azure service principal / OIDC for authentication

Pipeline steps:

- Install Terraform
- Set environment variables for credentials or use service connection
- Run init, plan, apply

---

#### 1.2 ARM Templates in Azure Pipelines

**ARM (Azure Resource Manager) templates** are JSON-based, Azure-native IaC.

Azure DevOps provides a built-in task:

- AzureResourceManagerTemplateDeployment@3

This task can:

- Deploy ARM templates to:
    - Resource group
    - Subscription
    - Management group
    - Tenant
- Support parameters & incremental/complete mode

Typical usage:

```
- task: AzureResourceManagerTemplateDeployment@3
  displayName: 'Deploy ARM template'
  inputs:
    deploymentScope: 'Resource Group'
    azureResourceManagerConnection: 'MyServiceConnection'
    subscriptionId: '$(subscriptionId)'
    action: 'Create Or Update Resource Group'
    resourceGroupName: 'rg-demo'
    location: 'Central India'
    templateLocation: 'Linked artifact'
    csmFile: 'infra/main.json'
    csmParametersFile: 'infra/parameters.json'
    deploymentMode: 'Incremental'
```

---

**1.3 Bicep in Azure Pipelines**

**Bicep** is a higher-level language that compiles to ARM.

You can deploy Bicep via:

1. **Azure CLI task** with az deployment ... commands
2. **AzureResourceManagerTemplateDeployment task** after pre-compiling Bicep to ARM (optional)

Example using Azure CLI task:

```
- task: AzureCLI@2
  displayName: 'Deploy Bicep file'
```

```
  inputs:
    azureSubscription: 'MyServiceConnection'
    scriptType: 'bash'
    scriptLocation: 'inlineScript'
    inlineScript: |
      az group create -n rg-demo -l centralindia
      az deployment group create \
        --resource-group rg-demo \
        --template-file infra/main.bicep \
        --parameters appName=myapp-demo
```

Bicep benefits:

- Cleaner syntax than ARM JSON
- Modules, type safety, IntelliSense in VS Code

---

**2. Service Connections (Azure RM / Service Principal)**

To let pipelines talk to Azure securely, Azure DevOps uses **service connections**.

---

**2.1 Azure Resource Manager Service Connection**

An **Azure RM service connection** uses:

- A **Service Principal** (App registration in Entra ID)
- With a defined **role** (typically Contributor)
- Scoped to:
    - A subscription, or
    - A resource group

Types:

1. **Service principal (automatic)**
    - Azure DevOps automatically creates SPN in your tenant
    - Recommended when you have admin rights and want easy setup
2. **Service principal (manual)**
    - You create SPN yourself (via portal/CLI)
    - Then supply **Client ID / Secret / Tenant / Subscription** in DevOps

- Useful when SPN is centrally managed by cloud team

3. **Managed Identity (for some scenarios)**
    - When using self-hosted agents running in Azure with managed identities
    - DevOps can use that identity instead of client secret

---

**2.2 Creating an Azure RM Service Connection**

1. In Azure DevOps, go to: **Project Settings → Service connections**
2. Click **New service connection**
3. Choose **Azure Resource Manager**
4. Choose **Service principal (automatic)** or **(manual)**
5. Select subscription and scope
6. Give a recognizable name (e.g. sc-azure-dev-terraform)
7. Grant access permission to all pipelines (if desired)

In YAML, you refer to it as:

azureSubscription: 'sc-azure-dev-terraform'

or

connectedServiceNameARM: 'sc-azure-dev-terraform'

depending on the task.

---

**2.3 Service Principal (Terraform Perspective)**

For Terraform, SPN can be passed via **environment variables**:

- ARM_CLIENT_ID
- ARM_CLIENT_SECRET
- ARM_TENANT_ID
- ARM_SUBSCRIPTION_ID

These are normally stored as **secret variables** or variable groups in Azure DevOps and mapped into pipeline environment.

---

**3. Lab: Provision Azure Resources via Terraform Pipeline**

Goal:
Use **Terraform + Azure DevOps YAML pipeline** to provision:

- Resource Group

- Storage Account

You can easily extend this later to App Service, VNets, etc.

---

**3.1 Terraform Files**

Assume repo structure:

```
infra/
  main.tf
  variables.tf
  backend.tf
```

**3.1.1 backend.tf (Remote backend in Azure Storage)**

```
terraform {
  backend "azurerm" {
    resource_group_name  = "rg-tfstate"
    storage_account_name = "sttfstate12345"
    container_name       = "tfstate"
    key                  = "demo.terraform.tfstate"
  }
}
```

Note: rg-tfstate, sttfstate12345, and tfstate should already exist (can be created manually or via a separate bootstrap step).

**3.1.2 main.tf**

```
terraform {
  required_version = ">= 1.5.0"
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0"
    }
  }
}
```

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

resource "azurerm_resource_group" "rg" {
  name     = var.resource_group_name
  location = var.location
}

resource "azurerm_storage_account" "sa" {
  name                     = var.storage_account_name
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
```

**3.1.3 variables.tf**

```
variable "resource_group_name" {
  type        = string
  description = "Name of the resource group"
}

variable "location" {
  type        = string
  description = "Azure region"
  default     = "Central India"
}

variable "storage_account_name" {
  type        = string
  description = "Storage account name (must be globally unique)"
```

}

Optionally create terraform.tfvars for local runs; pipeline can pass values via -var flags.

---

**3.2 Secure Variables for Terraform in Azure DevOps**

We'll authenticate Terraform via *ARM_ environment variables** set as pipeline variables (ideally in a variable group).

In **Library → Variable groups** (e.g. vg-terraform-azure):

- ARM_CLIENT_ID (secret)
- ARM_CLIENT_SECRET (secret)
- ARM_TENANT_ID (secret or non-secret)
- ARM_SUBSCRIPTION_ID (secret or non-secret)

These come from your Service Principal.

---

**3.3 YAML Pipeline – Terraform Plan & Apply**

Create azure-pipelines-terraform.yml in repo root:

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

variables:
- group: vg-terraform-azure   # Contains ARM_... env variables

stages:
# ------------------------------
# Stage 1: Terraform Validate & Plan
# ------------------------------
- stage: Terraform_Plan
```

```
displayName: 'Terraform - Validate & Plan'
jobs:
- job: Plan
  displayName: 'Terraform Plan'
  steps:
  - task: TerraformInstaller@1
    displayName: 'Install Terraform'
    inputs:
      terraformVersion: '1.6.0'

  - script: |
      cd infra
      terraform init
      terraform validate
    displayName: 'Terraform Init & Validate'
    env:
      ARM_CLIENT_ID:       $(ARM_CLIENT_ID)
      ARM_CLIENT_SECRET:    $(ARM_CLIENT_SECRET)
      ARM_TENANT_ID:       $(ARM_TENANT_ID)
      ARM_SUBSCRIPTION_ID:  $(ARM_SUBSCRIPTION_ID)

  - script: |
      cd infra
      terraform plan \
        -var "resource_group_name=rg-tf-demo" \
        -var "storage_account_name=sttf$(Build.BuildId)"
    displayName: 'Terraform Plan'
    env:
      ARM_CLIENT_ID:       $(ARM_CLIENT_ID)
      ARM_CLIENT_SECRET:    $(ARM_CLIENT_SECRET)
      ARM_TENANT_ID:       $(ARM_TENANT_ID)
```

```
      ARM_SUBSCRIPTION_ID:  $(ARM_SUBSCRIPTION_ID)


    - task: PublishBuildArtifacts@1
      displayName: 'Publish Plan (optional)'
      inputs:
        PathtoPublish: 'infra'
        ArtifactName: 'tf-infra'
        publishLocation: 'Container'


# --------------------------------
# Stage 2: Terraform Apply (with approval)
# --------------------------------
- stage: Terraform_Apply
  displayName: 'Terraform - Apply'
  dependsOn: Terraform_Plan
  condition: succeeded()
  jobs:
  - deployment: Apply
    displayName: 'Terraform Apply'
    environment: 'Terraform-Prod'       # environment can have approvals configured
    strategy:
      runOnce:
        deploy:
          steps:
          - task: TerraformInstaller@1
            displayName: 'Install Terraform'
            inputs:
              terraformVersion: '1.6.0'


          - script: |
              cd infra
```

```
  terraform init
  terraform apply -auto-approve \
    -var "resource_group_name=rg-tf-demo" \
    -var "storage_account_name=sttf$(Build.BuildId)"
displayName: 'Terraform Apply'
env:
  ARM_CLIENT_ID:       $(ARM_CLIENT_ID)
  ARM_CLIENT_SECRET:   $(ARM_CLIENT_SECRET)
  ARM_TENANT_ID:       $(ARM_TENANT_ID)
  ARM_SUBSCRIPTION_ID: $(ARM_SUBSCRIPTION_ID)
```