Modul 08

Infrastructure as Code

Lernziele

Nach diesem Modul kannst du:

- **Terraform-Konfigurationen in Azure Pipelines ausführen** - Den Plan/Apply-Workflow mit Remote State und manueller Freigabe umsetzen
- **Bicep-Templates in einer Pipeline deployen** - Azure-native Infrastruktur mit What-If-Analyse und Parameter-Dateien verwalten
- **Infrastructure Drift erkennen und behandeln** - Scheduled Pipelines mit Exit-Code-Auswertung und Benachrichtigungen einrichten

1. IaC-Grundlagen

Infrastructure as Code beschreibt Infrastruktur in Textdateien statt durch manuelle Konfiguration.

- **Deklarativ** - Du beschreibst den gewünschten Zustand, nicht die Schritte dorthin
- **Idempotent** - Mehrfaches Ausführen erzeugt dasselbe Ergebnis, keine doppelten Ressourcen
- **State** - Der aktuelle Zustand der Infrastruktur wird gespeichert und mit dem gewünschten Zustand verglichen

> **Kernprinzip:** Die Infrastruktur wird wie Anwendungscode behandelt -- versioniert, reviewt und automatisiert deployed.

1.1 Vorteile von IaC

Vorteil	Beschreibung
Versionierung	Infrastruktur-Änderungen im Git nachvollziehen
Reproduzierbarkeit	Identische Umgebungen per Knopfdruck erstellen
Automatisierung	Deployment über CI/CD-Pipelines
Konsistenz	Keine Konfigurationsdrift durch manuelle Änderungen
Review-Prozess	Pull Requests für Infrastruktur-Änderungen
Dokumentation	Code ist die Dokumentation der Infrastruktur

2. Terraform Grundlagen

Terraform von HashiCorp ist ein Cloud-agnostisches IaC-Tool.

- Infrastruktur wird in `.tf`-**Dateien** deklarativ beschrieben (HCL)
- Unterstützt **viele Provider**: Azure, AWS, GCP, Kubernetes, ...
- Verwaltet einen **State**, der den aktuellen Zustand speichert
- Arbeitet mit **Ressourcen**, **Variablen** und **Outputs**

```
resource "azurerm_resource_group" "app" {
  name     = "rg-${var.project_name}-${var.environment}"
  location = var.location
  tags     = { ManagedBy = "terraform" }
}
```

2.1 Terraform: init, plan, apply

Der Terraform-Workflow besteht aus drei Schritten:

```
# 1. Provider herunterladen, Backend initialisieren
terraform init

# 2. Änderungen vorschauen (Dry Run)
terraform plan -var-file=dev.tfvars -out=tfplan

# 3. Änderungen anwenden
terraform apply tfplan
```

- **init** - Einmalig pro Arbeitsverzeichnis, lädt Provider-Plugins
- **plan** - Vergleicht Code mit State, zeigt geplante Änderungen
- **apply** - Führt die geplanten Änderungen durch

2.2 Remote State (Azure Storage Backend)

Der Terraform State muss **zentral gespeichert** werden, damit Pipeline und Team denselben State verwenden.

```
terraform {
  backend "azurerm" {
    resource_group_name  = "rg-pipeline-training"
    storage_account_name = "stterraformstate"
    container_name       = "tfstate"
    key                  = "training.terraform.tfstate"
  }
}
```

Vorteile des Remote State:

- **State Locking** - Verhindert parallele Änderungen
- **Verschlüsselung** - State enthält sensible Daten
- **Geteilter Zugriff** - Team und Pipeline nutzen denselben State

2.3 Service Principal für Azure

Terraform benötigt einen **Service Principal** für die Authentifizierung in der Pipeline:

- In Azure DevOps als **Service Connection** konfiguriert
- Der `TerraformTaskV4` nutzt `environmentServiceNameAzureRM`
- Berechtigungen: **Contributor** auf der Ziel-Subscription

> **Wichtig:** Die Terraform-Extension von Microsoft DevLabs muss in der Azure DevOps Organisation installiert sein.

2.4 Terraform Pipeline

```yaml
stages:
  - stage: Plan
    jobs:
      - job: TerraformPlan
        steps:
          - task: TerraformInstaller@1
            inputs:
              terraformVersion: 'latest'
          - task: TerraformTaskV4@4
            displayName: 'Terraform Init'
            inputs:
              provider: 'azurerm'
              command: 'init'
              backendServiceArm: '$(azureSubscription)'
              backendAzureRmResourceGroupName: '$(stateResourceGroup)'
              backendAzureRmStorageAccountName: '$(storageAccount)'
              backendAzureRmContainerName: '$(stateContainer)'
              backendAzureRmKey: 'training.terraform.tfstate'
          - task: TerraformTaskV4@4
            displayName: 'Terraform Plan'
            inputs:
              provider: 'azurerm'
              command: 'plan'
              commandOptions: '-var-file=dev.tfvars -out=tfplan'
              environmentServiceNameAzureRM: '$(azureSubscription)'
```

3. Bicep: Azure-native DSL

Bicep ist Microsofts eigene IaC-Sprache für Azure.

- **Abstraktion über ARM-Templates** -- lesbarere Syntax
- **In Azure CLI integriert** -- kein separates Tool nötig
- **Automatische Abhängigkeitserkennung** zwischen Ressourcen
- **Modulare Struktur** für Wiederverwendbarkeit

```
resource webApp 'Microsoft.Web/sites@2023-01-01' = {
  name: '${projectName}-${environment}-${uniqueSuffix}'
  location: location
  properties: {
    serverFarmId: appServicePlan.id
  }
}
```

3.1 Bicep vs ARM Templates

Aspekt	Bicep	ARM Template (JSON)
Syntax	Kurz und lesbar	Verbose JSON
Abhängigkeiten	Automatisch erkannt	Manuell mit `dependsOn`
State-Management	Keins nötig (Azure verwaltet)	Keins nötig
Cloud-Support	Nur Azure	Nur Azure
Tooling	In Azure CLI integriert	Separater Validator nötig
Module	`module` Keyword	Linked/Nested Templates
Typ-Sicherheit	Ja (Compile-Time-Checks)	Nein

> **Empfehlung:** Für reine Azure-Projekte ist Bicep die bevorzugte Wahl. Für Multi-Cloud-Projekte eignet sich Terraform besser.

3.2 Bicep: What-If und Parameter-Dateien

What-If zeigt eine Vorschau der Änderungen (wie `terraform plan`):

```
az deployment group what-if \
  --resource-group rg-pipeline-training \
  --template-file bicep/main.bicep \
  --parameters bicep/dev.parameters.json
```

Parameter-Dateien trennen Konfiguration von Template:

```
{
  "parameters": {
    "projectName": { "value": "training-app" },
    "environment": { "value": "dev" },
    "uniqueSuffix": { "value": "mmu" }
  }
}
```

3.3 Bicep Pipeline

```yaml
stages:
  - stage: WhatIf
    jobs:
      - job: WhatIfAnalysis
        steps:
          - task: AzureCLI@2
            displayName: 'What-If Deployment'
            inputs:
              azureSubscription: '$(azureSubscription)'
              scriptType: 'bash'
              scriptLocation: 'inlineScript'
              inlineScript: |
                az deployment group what-if \
                  --resource-group $(resourceGroup) \
                  --template-file bicep/main.bicep \
                  --parameters bicep/dev.parameters.json
  - stage: Deploy
    jobs:
      - deployment: DeployBicep
        environment: 'dev'
        strategy:
          runOnce:
            deploy:
              steps:
                - task: AzureCLI@2
                  inputs:
                    azureSubscription: '$(azureSubscription)'
                    scriptType: 'bash'
                    scriptLocation: 'inlineScript'
                    inlineScript: |
                      az deployment group create \
                        --resource-group $(resourceGroup) \
                        --template-file bicep/main.bicep \
                        --parameters bicep/dev.parameters.json
```

4. Drift Detection Konzept

Infrastructure Drift entsteht, wenn die tatsächliche Infrastruktur von der IaC-Definition abweicht.

Typische Ursachen:

- Manuelle Änderungen im Azure Portal
- Änderungen durch andere Pipelines oder Skripte
- Automatische Updates durch Azure

Warum ist Drift gefährlich?

- Unerwartete Fehler bei Deployments
- Vertrauen in IaC-Definitionen wird untergraben
- Sicherheitskonfigurationen werden umgangen

4.1 Scheduled Pipeline für Drift Detection

Drift-Checks laufen als **Scheduled Pipeline** -- ohne CI-Trigger:

```yaml
trigger: none  # Kein CI-Trigger

schedules:
  - cron: '0 7 * * Mon-Fri'
    displayName: 'Täglicher Drift Check (07:00 UTC)'
    branches:
      include:
        - main
    always: true  # Auch ohne Code-Änderungen ausführen
```

- `trigger: none` - Pipeline startet nicht bei Code-Änderungen
- `always: true` - Läuft auch wenn kein neuer Commit vorliegt
- **Empfehlung:** Täglich an Werktagen, morgens vor Arbeitsbeginn

4.2 Exit-Codes und Benachrichtigungen

Terraform nutzt `plan -detailed-exitcode` für Drift Detection:

Exit Code	Bedeutung	Aktion
0	Kein Drift	Alles OK, keine Aktion
1	Fehler	Pipeline fehlgeschlagen
2	Drift erkannt	Alert senden, Review nötig

Bicep nutzt `what-if` zur Erkennung:

Ergebnis	Bedeutung
`NoChange`	Kein Drift -- alles synchron
`Modify`	Einstellungen geändert -- Drift!
`Create`	Ressource fehlt (manuell gelöscht)
`Delete`	Ressource existiert, nicht im Code

Labs

- **Lab 16:** Terraform mit Azure Pipelines
- **Lab 17:** Bicep-Deployments
- **Lab 18:** Infrastructure Drift Detection