Terraform Workspaces

Terraform workspaces provide a way to manage multiple environments (like **dev, staging, and prod**) using the same configuration without duplicating code.

Why Use Terraform Workspaces?

- Manage multiple environments without copying .tf files.
- Keep a single state file per workspace.
- Easy to switch between different environments.

How Terraform Workspaces Work

Terraform maintains a **default workspace** but allows you to create and switch between multiple workspaces.

Commands:
Check Current Workspace
terraform workspace show

List All Workspaces
terraform workspace list

Create a New Workspace
terraform workspace new dev

Switch to an Existing Workspace
terraform workspace select dev

Delete a Workspace
terraform workspace delete dev

Example: Using Workspaces in Terraform

main.tf

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"

  tags = {
    Name        = "Terraform-${terraform.workspace}"  # Workspace name in
resource name
    Environment = terraform.workspace
  }
}
```

Behavior

- If you're in the dev workspace, the instance name will be **Terraform-dev**.
- If you're in the prod workspace, the instance name will be **Terraform-prod**.

Best Practices

- Use workspaces when managing multiple environments with **the same configuration**.

- For completely different configurations, use **separate folders/modules** instead.
- Combine workspaces with **Terraform backend (S3, GCS, etc.)** for better state management.

Step-by-Step Guide to Using Terraform Workspaces

Let's create a Terraform project that provisions an **AWS EC2 instance** in different environments (**dev, staging, prod**) using Terraform workspaces.

🛠️ Step 1: Initialize a Terraform Project

Create a new directory for your Terraform project:
mkdir terraform-workspace-demo && cd terraform-workspace-demo

Create a main.tf file:
touch main.tf

📝 Step 2: Define Resources in main.tf

Open main.tf and add the following Terraform configuration:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-12345678"  # Replace with a valid AMI ID
  instance_type = "t2.micro"

  tags = {
```

```
    Name        = "Terraform-${terraform.workspace}"  # Set name dynamically
based on workspace
    Environment = terraform.workspace
  }
}
```

🚀 Step 3: Initialize Terraform

Run the following command to initialize Terraform:

```
terraform init
```

🌍 Step 4: Create & Manage Workspaces

Check the default workspace:
terraform workspace show
Output: default

Create a new workspace (dev):
terraform workspace new dev
Output: Created and switched to workspace "dev"!

Create another workspace (staging):
terraform workspace new staging
Output: Created and switched to workspace "staging"!

Switch between workspaces:
terraform workspace select dev

Output: Switched to workspace "dev".

List all workspaces:
terraform workspace list
Output:
  default
* dev
  staging

📌 Step 5: Apply Configuration Based on Workspace

Now, apply Terraform configuration for the current workspace (dev):

terraform apply -auto-approve

- This will create an **EC2 instance** with a name like Terraform-dev.

Switch to staging and apply:

terraform workspace select staging
terraform apply -auto-approve

- This will create an **EC2 instance** with a name like Terraform-staging.

🗑️ Step 6: Destroy Resources When Not Needed

To delete resources from a specific workspace:

terraform destroy -auto-approve

To delete a workspace:

terraform workspace select default
terraform workspace delete dev

🎯 Summary

- Terraform workspaces allow **managing multiple environments without duplicating code**.
- The terraform.workspace variable dynamically sets environment-specific values.
- Use terraform workspace new <name> to create workspaces.
- Use terraform workspace select <name> to switch between them.

📂 Real-World Terraform Project Structure Using Workspaces

📌 Project Folder Structure

terraform-workspace-demo/
├── environments/
│ ├── dev.tfvars
│ ├── staging.tfvars
│ ├── prod.tfvars
├── main.tf
├── variables.tf
├── outputs.tf
├── backend.tf # Optional for remote state management
├── README.md

📝 main.tf - Define AWS Resources

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = var.ami
  instance_type = var.instance_type

  tags = {
    Name        = "Terraform-${terraform.workspace}"
    Environment = terraform.workspace
  }
}
```

📌 variables.tf - Define Variables

```
variable "ami" {
  description = "AMI ID for EC2 instance"
  type        = string
}

variable "instance_type" {
  description = "Instance type"
  type        = string
}
```

📌 environments/dev.tfvars - Dev Environment Variables

```
ami = "ami-12345678"
```

instance_type = "t2.micro"

📌 environments/staging.tfvars - Staging Environment Variables

ami = "ami-87654321"
instance_type = "t3.small"

📌 environments/prod.tfvars - Production Environment Variables

ami = "ami-11223344"
instance_type = "t3.medium"

📌 backend.tf (Optional) - Remote State Management

To store Terraform state remotely (e.g., in S3):

```
terraform {
  backend "s3" {
    bucket = "my-terraform-state-bucket"
    key    = "terraform-${terraform.workspace}.tfstate"
    region = "us-east-1"
  }
}
```

🚀 Step-by-Step Execution

1️⃣ Initialize Terraform

terraform init

2️⃣ Create a new workspace (e.g., dev)

terraform workspace new dev

3️⃣ Select workspace before applying changes

terraform workspace select dev

4️⃣ Apply configuration for dev environment

terraform apply -var-file=environments/dev.tfvars -auto-approve

5️⃣ Switch to staging environment and apply

terraform workspace select staging
terraform apply -var-file=environments/staging.tfvars -auto-approve

6️⃣ Switch to prod environment and apply

terraform workspace select prod
terraform apply -var-file=environments/prod.tfvars -auto-approve

---

🗑️ Cleanup

To destroy resources in a workspace:
terraform destroy -var-file=environments/dev.tfvars -auto-approve

To delete a workspace:
terraform workspace select default
terraform workspace delete dev

🎯 Key Takeaways

✅ **Workspaces help manage multiple environments easily**
✅ **Use terraform.workspace for dynamic naming**
✅ **Store environment-specific configurations in .tfvars files**
✅ **Use S3 for remote state storage in a multi-team setup**

🚀 CI/CD Pipeline with Terraform Workspaces

A Terraform CI/CD pipeline automates the infrastructure deployment for different environments (**dev, staging, prod**) using workspaces.

📌 Overview

We'll create a **GitHub Actions Pipeline** that:

1. **Checks out the Terraform code**
2. **Initializes Terraform**
3. **Selects the correct workspace**
4. **Applies Terraform configuration based on the environment**

📂 Folder Structure

```
terraform-workspace-demo/
├── .github/workflows/
│   ├── terraform-ci.yml  # GitHub Actions workflow
├── environments/
│   ├── dev.tfvars
│   ├── staging.tfvars
│   ├── prod.tfvars
├── main.tf
├── variables.tf
```

```
├── outputs.tf
├── backend.tf  # Optional for remote state
├── README.md
```

📝 terraform-ci.yml - GitHub Actions Workflow

```
name: Terraform CI/CD

on:
  push:
    branches:
      - main  # Trigger on push to main branch

jobs:
  terraform:
    runs-on: ubuntu-latest

    env:
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY
}}
      TF_WORKSPACE: ${{ github.event.inputs.workspace }}

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v3
        with:
          terraform_version: 1.5.0
```

```
    - name: Initialize Terraform
      run: terraform init

    - name: Select Terraform Workspace
      run: |
        terraform workspace select $TF_WORKSPACE || terraform workspace new $TF_WORKSPACE

    - name: Apply Terraform Configuration
      run: terraform apply -var-file=environments/${TF_WORKSPACE}.tfvars -auto-approve
```

🔑 Secrets Required (GitHub Settings → Secrets and Variables)

Secret Name	Value
AWS_ACCESS_KEY_ID	Your AWS Access Key
AWS_SECRET_ACCESS_KEY	Your AWS Secret Key

🚀 Running the CI/CD Pipeline

1️⃣ Push code to GitHub

```
git add .
git commit -m "Terraform workspace automation"
git push origin main
```

2️⃣ Trigger the workflow manually (GitHub Actions → terraform-ci.yml)

- Select the **workspace (dev, staging, or prod)**
- Click **Run workflow**

📌 What Happens?

✅ **CI/CD pipeline initializes Terraform**
✅ **Selects or creates the correct workspace**
✅ **Applies the Terraform configuration for the specified environment**

🗑️ Cleanup

Destroy resources via CI/CD:
yaml

```
- name: Destroy Terraform Resources
  run: terraform destroy -var-file=environments/${TF_WORKSPACE}.tfvars
-auto-approve
```

🎯 Key Takeaways

✅ **Automates Terraform deployments using GitHub Actions**
✅ **Uses Terraform workspaces to manage multiple environments**
✅ **Secure AWS credentials using GitHub Secrets**

🚀 Terraform Workspaces with Jenkins CI/CD Pipeline

This Jenkins pipeline automates infrastructure deployment for multiple environments (**dev, staging, prod**) using Terraform workspaces.

📂 Folder Structure

terraform-workspace-demo/

```
├── environments/
│   ├── dev.tfvars
│   ├── staging.tfvars
│   ├── prod.tfvars
├── main.tf
├── variables.tf
├── outputs.tf
├── backend.tf  # Optional for remote state
├── Jenkinsfile
├── README.md
```

📝 Jenkinsfile - Terraform Pipeline

```
groovy

pipeline {
    agent any

    parameters {
        choice(name: 'WORKSPACE', choices: ['dev', 'staging', 'prod'], description:
'Select the Terraform workspace')
    }

    environment {
        AWS_ACCESS_KEY_ID = credentials('AWS_ACCESS_KEY_ID')
        AWS_SECRET_ACCESS_KEY =
credentials('AWS_SECRET_ACCESS_KEY')
        TF_WORKSPACE = "${params.WORKSPACE}"
    }

    stages {
        stage('Checkout Code') {
            steps {
```

```
                git branch: 'main', url:
'https://github.com/your-repo/terraform-workspace-demo.git'
            }
        }

        stage('Setup Terraform') {
            steps {
                sh 'terraform init'
            }
        }

        stage('Select or Create Workspace') {
            steps {
                sh """
                terraform workspace select ${TF_WORKSPACE} || terraform
workspace new ${TF_WORKSPACE}
                """
            }
        }

        stage('Apply Terraform Configuration') {
            steps {
                sh "terraform apply -var-file=environments/${TF_WORKSPACE}.tfvars
-auto-approve"
            }
        }
    }

    post {
        success {
            echo "Terraform applied successfully for workspace:
${TF_WORKSPACE}"
        }
        failure {
            echo "Terraform failed for workspace: ${TF_WORKSPACE}"
```

```
        }
    }
}
```

🔑 Jenkins Setup

1. **Create AWS Credentials in Jenkins**
    - Go to **Jenkins Dashboard → Manage Jenkins → Manage Credentials**
    - Add AWS credentials with ID **AWS_ACCESS_KEY_ID**
2. **Install Terraform Plugin (Optional)**
    - Manage Jenkins → Manage Plugins → Install **Terraform Plugin**

🚀 Running the Pipeline

1️⃣ Go to Jenkins → New Item → Pipeline
2️⃣ Select "This project is parameterized"

- Add **Choice Parameter** for WORKSPACE
- Options: dev, staging, prod

3️⃣ Run the Pipeline

- Select **Workspace (e.g., dev)**
- Click **Build Now**

🗑️ Cleanup

To destroy resources, modify the Jenkinsfile:

groovy

```
stage('Destroy Terraform') {
```

```
    steps {
        sh "terraform destroy -var-file=environments/${TF_WORKSPACE}.tfvars
-auto-approve"
    }
}
```

🎯 Key Takeaways

✅ **Automates Terraform deployments using Jenkins**
✅ **Uses parameterized builds for selecting environments**
✅ **Securely manages AWS credentials using Jenkins Secrets**

🚀 Integrating Terraform Workspaces with Kubernetes (EKS)

This guide will show how to **provision an AWS EKS cluster** using Terraform workspaces and deploy an application using **Kubernetes manifests**.

📂 Folder Structure

```
terraform-k8s-integration/
├── environments/          # ✅ Holds environment-specific variable files
│   ├── dev.tfvars
│   ├── staging.tfvars
│   ├── prod.tfvars
├── modules/               # ✅ Reusable Terraform modules
│   ├── eks/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
```

```
├── k8s-manifests/          # ✅ Kubernetes deployment files
│   ├── deployment.yaml
│   ├── service.yaml
├── main.tf                 # ✅ Calls modules & sets up infrastructure
├── variables.tf            # ✅ Global variables
├── outputs.tf              # ✅ Outputs
├── backend.tf              # ✅ Remote state config (S3, DynamoDB)
├── Jenkinsfile             # ✅ CI/CD pipeline for automation
├── README.md               # ✅ Documentation
```

📝 main.tf - Provision EKS Cluster Using Terraform

```
# Define Terraform backend for remote state storage
terraform {
  required_version = ">= 1.5.0"

  backend "s3" {
    bucket = "my-terraform-state-bucket"
    key    = "terraform-${terraform.workspace}.tfstate"
    region = "us-east-1"
    dynamodb_table = "terraform-lock"
  }
}

# Define the AWS provider
provider "aws" {
  region = "us-east-1"
}

# Call the EKS module to create the cluster
module "eks" {
  source  = "./modules/eks"
```

```
  cluster_name = var.cluster_name
  node_count = var.node_count
}
```

📌 variables.tf - Global Variables

```
variable "cluster_name" {
  description = "EKS cluster name"
  type        = string
}

variable "node_count" {
  description = "Number of worker nodes"
  type        = number
}
```

📌 environments/dev.tfvars - Dev-Specific Config

```
# Dev environment-specific values
cluster_name = "eks-dev"
node_count   = 2
```

📌 environments/staging.tfvars - Staging Config

```
# Staging environment-specific values
cluster_name = "eks-staging"
node_count   = 3
```

📌 environments/prod.tfvars - Production Config

```
# Production environment-specific values
```

```
cluster_name = "eks-prod"
node_count   = 5
```

📝 modules/eks/main.tf - EKS Cluster Module

```
resource "aws_eks_cluster" "eks" {
  name     = var.cluster_name
  role_arn = aws_iam_role.eks_role.arn

  vpc_config {
    subnet_ids = [aws_subnet.subnet1.id, aws_subnet.subnet2.id]
  }

  tags = {
    Name = "EKS-${terraform.workspace}"
  }
}

resource "aws_eks_node_group" "worker_nodes" {
  cluster_name  = aws_eks_cluster.eks.name
  node_role_arn = aws_iam_role.node_role.arn
  subnet_ids    = [aws_subnet.subnet1.id, aws_subnet.subnet2.id]
  scaling_config {
    desired_size = var.node_count
    max_size     = 5
    min_size     = 1
  }

  tags = {
    Name = "EKS-Workers-${terraform.workspace}"
  }
}
```

📝 k8s-manifests/deployment.yaml - Kubernetes App Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: nginx
          ports:
            - containerPort: 80
```

📝 k8s-manifests/service.yaml - Kubernetes Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
```

```
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer
```

🚀 Running Terraform and Deploying Kubernetes Workloads

1️⃣ Initialize Terraform

```
terraform init
```

2️⃣ Create & Select Workspace

```
terraform workspace new dev  # Create a new workspace for development
terraform workspace select dev  # Switch to the dev workspace
```

3️⃣ Apply Terraform to Create EKS Cluster

```
terraform apply -var-file=environments/dev.tfvars -auto-approve
```

4️⃣ Configure kubectl for EKS

```
aws eks update-kubeconfig --name eks-dev --region us-east-1
```

5️⃣ Deploy Kubernetes Application

```
kubectl apply -f k8s-manifests/deployment.yaml
kubectl apply -f k8s-manifests/service.yaml
```

📝 Jenkinsfile - Automating Terraform + Kubernetes Deployments

```
pipeline {
    agent any

    parameters {
        choice(name: 'WORKSPACE', choices: ['dev', 'staging', 'prod'], description: 'Select the Terraform workspace')
    }

    environment {
        AWS_ACCESS_KEY_ID = credentials('AWS_ACCESS_KEY_ID')  // Secure AWS credentials
        AWS_SECRET_ACCESS_KEY = credentials('AWS_SECRET_ACCESS_KEY')
        TF_WORKSPACE = "${params.WORKSPACE}"  // Assign workspace parameter
    }

    stages {
        stage('Checkout Code') {
            steps {
                git branch: 'main', url: 'https://github.com/your-repo/terraform-k8s-integration.git'  // Pull latest code from repo
            }
        }

        stage('Setup Terraform') {
            steps {
                sh 'terraform init'  // Initialize Terraform
            }
        }
```

```
        stage('Select or Create Workspace') {
            steps {
                sh """
                terraform workspace select ${TF_WORKSPACE} || terraform
workspace new ${TF_WORKSPACE}  // Switch or create workspace
                """
            }
        }

        stage('Apply Terraform Configuration') {
            steps {
                sh "terraform apply -var-file=environments/${TF_WORKSPACE}.tfvars
-auto-approve"  // Apply Terraform for selected workspace
            }
        }

        stage('Configure kubectl for EKS') {
            steps {
                sh "aws eks update-kubeconfig --name eks-${TF_WORKSPACE}
--region us-east-1"  // Configure kubectl
            }
        }

        stage('Deploy Kubernetes App') {
            steps {
                sh "kubectl apply -f k8s-manifests/deployment.yaml"  // Deploy app
                sh "kubectl apply -f k8s-manifests/service.yaml"  // Deploy service
            }
        }
    }

    post {
        success {
            echo "Terraform & Kubernetes deployment successful for workspace:
${TF_WORKSPACE}"
```

```
        }
        failure {
            echo "Terraform or Kubernetes deployment failed for workspace: ${TF_WORKSPACE}"
        }
    }
}
```

🎯 Best Practices

✅ **Use Terraform workspaces to manage multiple Kubernetes clusters**
✅ **Store Terraform state remotely (S3 + DynamoDB) for multi-user access**
✅ **Automate deployments with Jenkins for Terraform & Kubernetes**
✅ **Use kubectl to deploy workloads after provisioning EKS**