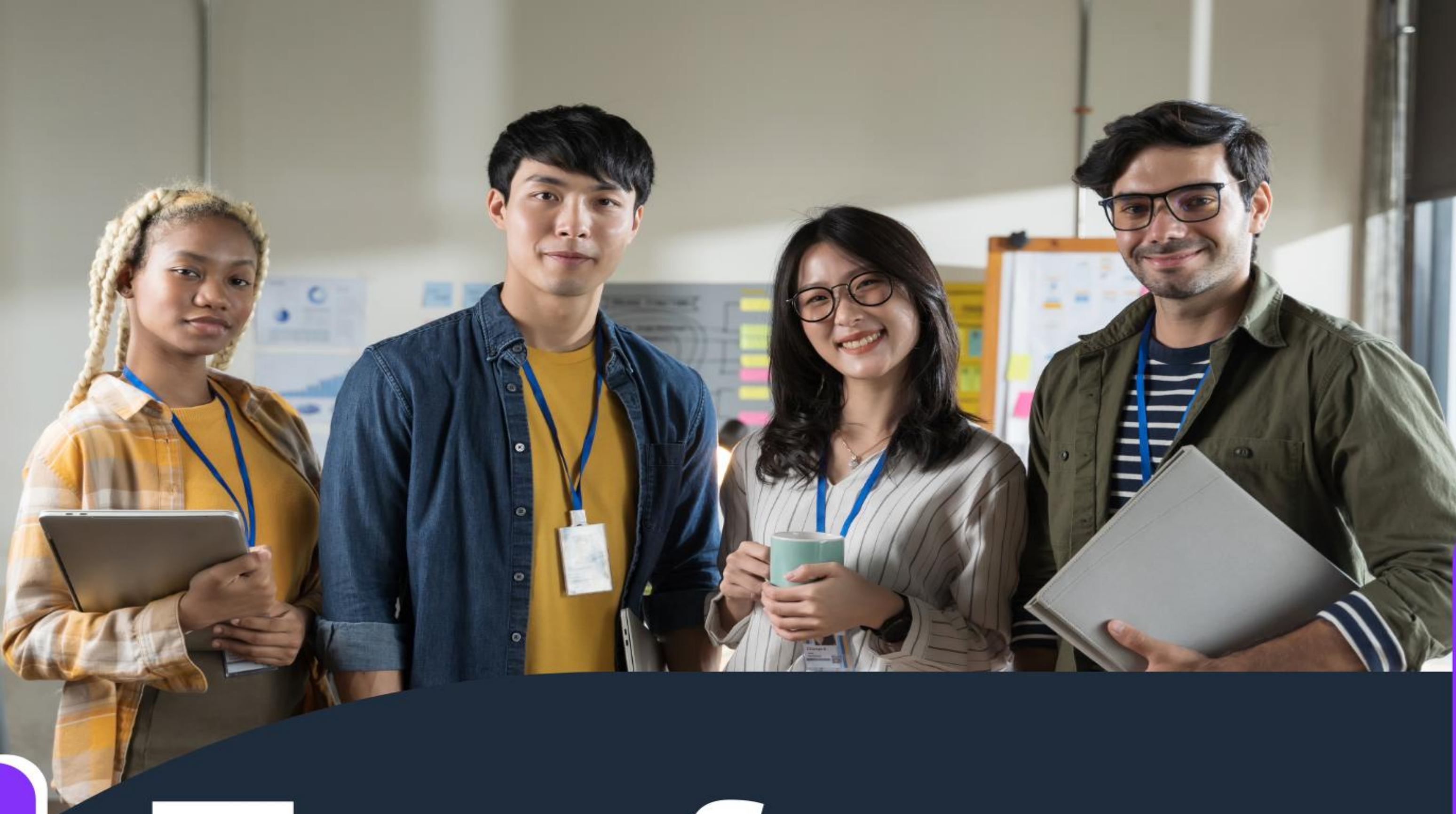

Terraform Projects

Top 5 Terraform Projects to Master Cloud Infrastructure Automation

2025

Devops Shack

www.devopsshack.com

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

Top 5 Terraform Projects to Master Cloud Infrastructure Automation

Table of Contents

Introduction

Terraform, developed by HashiCorp, is one of the most powerful and widely used tools in the world of Infrastructure as Code (IaC). It allows engineers, developers, and cloud architects to define and provision infrastructure resources in a consistent, repeatable, and automated manner. With Terraform, infrastructure management becomes simpler, scalable, and free from the pitfalls of manual configuration, making it a cornerstone for cloud automation and DevOps practices.

In today’s dynamic cloud-driven environment, mastering Terraform has become a must-have skill for professionals. The ability to write declarative configuration files and manage infrastructure across major cloud providers like AWS, Azure, and Google Cloud is invaluable for anyone aiming to enhance their expertise in cloud computing and DevOps.

This guide introduces five practical and real-world projects that showcase Terraform's capabilities and highlight how it can be used to automate different aspects of cloud infrastructure. Each project has been carefully designed to help you gain hands-on experience, from deploying high-availability web applications to setting up CI/CD pipelines and building serverless applications. By working through these projects, you'll learn how to:

- Deploy secure, scalable, and fault-tolerant web applications.
- Automate Kubernetes cluster provisioning and management.
- Build and automate a complete CI/CD pipeline.
- Leverage serverless technologies like AWS Lambda and DynamoDB.
- Integrate infrastructure automation seamlessly into your workflow.

Whether you are a beginner looking to kickstart your journey in cloud automation or an experienced professional wanting to deepen your expertise, this document serves as a practical, hands-on resource. Each project comes with detailed implementation steps, helping you understand the core concepts while applying them to real-world scenarios. So, let’s dive into the world of Terraform and explore how it can transform the way you manage and automate your infrastructure!

Project 1: Deploy a High Availability (HA) Web Application on AWS

This project demonstrates how to deploy a highly available web application on AWS using Terraform. The infrastructure includes a Virtual Private Cloud (VPC), subnets, an internet gateway, a route table, EC2 instances, an application load balancer (ALB), and an auto-scaling group. The goal is to ensure fault tolerance and scalability for the web application.

Implementation Steps

Step 1: Install and Configure Terraform

1. **Install Terraform**: Download Terraform from the official website and install it on your local machine.
2. **Set up AWS CLI**: Configure the AWS CLI with your credentials using the following command:

```
aws configure
```

Provide your AWS Access Key, Secret Key, default region (e.g., us-east-1), and default output format.

3. **Create a Working Directory**: Create a folder for your project, e.g., terraform-ha-web-app.

Step 2: Initialize Terraform Project

1. Inside the project folder, create a file named main.tf and add the AWS provider configuration:

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

2. Run the following command to initialize Terraform and download the AWS provider plugin:

```
terraform init
```

Step 3: Create a VPC

A Virtual Private Cloud (VPC) isolates your resources and provides networking infrastructure.

1. Define a VPC in a new file called vpc.tf:

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "terraform-vpc"
  }
}
```

2. This configuration creates a VPC with the CIDR block 10.0.0.0/16.

Step 4: Create Public and Private Subnets

Subnets divide your VPC into smaller networks. Public subnets allow access to the internet, while private subnets do not.

1. Add the subnet configurations to vpc.tf:

```
resource "aws_subnet" "public" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"
  map_public_ip_on_launch = true
  availability_zone = "us-east-1a"
  tags = {
    Name = "public-subnet"
  }
}

resource "aws_subnet" "private" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.2.0/24"
  availability_zone = "us-east-1b"
  tags = {
    Name = "private-subnet"
  }
}
```

2. The public subnet is configured to assign public IPs to instances automatically.

Step 5: Add an Internet Gateway and Route Table

An internet gateway allows internet traffic to flow to resources in the public subnet.

1. In vpc.tf, add the following resources:

```
resource "aws_internet_gateway" "main" {
  vpc_id = aws_vpc.main.id
  tags = {
    Name = "terraform-igw"
  }
}

resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.main.id
  }
}

resource "aws_route_table_association" "public" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public.id
}
```

2. This configuration sets up internet access for resources in the public subnet.

Step 6: Launch EC2 Instances

Create web server instances to host your application.

1. Create a new file ec2.tf and define an EC2 instance:

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"  # Amazon Linux 2 AMI
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.public.id
  key_name      = "your-key-pair"

  tags = {
    Name = "web-server"
  }
}
```

2. Ensure you have an existing key pair in your AWS account for SSH access.

Step 7: Set Up an Application Load Balancer

An ALB distributes incoming traffic across multiple instances for high availability.

1. Create a new file alb.tf and define the ALB:

```
resource "aws_lb" "app" {
  name               = "terraform-alb"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.alb_sg.id]
  subnets            = [aws_subnet.public.id]

  tags = {
    Name = "terraform-alb"
  }
}

resource "aws_lb_target_group" "web_tg" {
  name     = "web-target-group"
  port     = 80
  protocol = "HTTP"
  vpc_id   = aws_vpc.main.id
}

resource "aws_lb_listener" "web_listener" {
  load_balancer_arn = aws_lb.app.arn
  port              = 80
  protocol          = "HTTP"
  default_action {
    type = "forward"
    target_group_arn = aws_lb_target_group.web_tg.arn
  }
}
```

Step 8: Configure Auto-Scaling

Set up an auto-scaling group to ensure the application scales based on demand.

1. In autoscaling.tf, add the following:

```
resource "aws_launch_configuration" "web" {
  name          = "web-lc"
  image_id      = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  key_name      = "your-key-pair"

  lifecycle {
    create_before_destroy = true
  }
}

resource "aws_autoscaling_group" "web" {
  launch_configuration = aws_launch_configuration.web.id
  min_size             = 1
  max_size             = 3
  desired_capacity     = 2
  vpc_zone_identifier  = [aws_subnet.public.id]

  tag {
    key                 = "Name"
    value               = "web-instance"
    propagate_at_launch = true
  }
}
```

2. This configuration ensures the application can handle varying traffic loads.

Step 9: Apply the Terraform Configuration

1. Initialize Terraform again:

```
terraform init
```

2. Validate the configuration to ensure there are no syntax errors:

```
terraform validate
```

3. Preview the infrastructure changes:

```
terraform plan
```

4. Deploy the infrastructure:

```
terraform apply
```

5. Confirm the deployment when prompted.

Outcome

- **Infrastructure Components**:
  - A VPC with public and private subnets.
  - An internet gateway for public subnet access.
  - EC2 instances running in an auto-scaling group.
  - An application load balancer routing traffic to the instances.
- **Access**:
  - The web application is accessible via the ALB's DNS name.
  - Traffic is distributed across instances to ensure high availability.

Project 2: Deploy a Secure Multi-Tier Web Application on AWS

This project focuses on deploying a secure multi-tier web application architecture on AWS. The architecture consists of a public-facing web tier, an internal application tier, and a database tier hosted in private subnets. A bastion host is used for secure access to private resources.

Implementation Steps

Step 1: Set Up Terraform

1. Install Terraform on your local machine.
2. Create a directory for your project, e.g., terraform-multi-tier-app.

Step 2: Define the AWS Provider

1. Create a main.tf file and configure the AWS provider:

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

2. Run the initialization command:

```
terraform init
```

Step 3: Create a VPC

A Virtual Private Cloud (VPC) provides isolated networking for your application.

1. In vpc.tf, define the VPC:

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "multi-tier-vpc"
  }
}
```

Step 4: Create Subnets

1. Add public and private subnets for each tier in vpc.tf:

```
resource "aws_subnet" "public" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"
  map_public_ip_on_launch = true
  availability_zone = "us-east-1a"
  tags = {
    Name = "public-subnet"
  }
}

resource "aws_subnet" "app" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.2.0/24"
  availability_zone = "us-east-1b"
  tags = {
    Name = "app-subnet"
  }
}

resource "aws_subnet" "db" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.3.0/24"
  availability_zone = "us-east-1c"
  tags = {
    Name = "db-subnet"
  }
}
```

Step 5: Add an Internet Gateway

An internet gateway allows access to the public-facing web tier.

1. In vpc.tf, add the gateway and route table:

```
resource "aws_internet_gateway" "main" {
  vpc_id = aws_vpc.main.id
  tags = {
    Name = "multi-tier-igw"
  }
}

resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.main.id
  }
}

resource "aws_route_table_association" "public" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public.id
}
```

Step 6: Set Up Security Groups

Define security groups for each tier to control access.

1. Create a security_groups.tf file:

```
resource "aws_security_group" "web_sg" {
  vpc_id = aws_vpc.main.id
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "web-sg"
  }
}
```

```
resource "aws_security_group" "app_sg" {
 vpc_id = aws_vpc.main.id
 ingress {
  from_port   = 8080
  to_port     = 8080
  protocol    = "tcp"
  security_groups = [aws_security_group.web_sg.id]
 }

 egress {
  from_port   = 0
  to_port     = 0
  protocol    = "-1"
  cidr_blocks = ["0.0.0.0/0"]
 }

 tags = {
  Name = "app-sg"
 }
}

resource "aws_security_group" "db_sg" {
 vpc_id = aws_vpc.main.id
 ingress {
  from_port   = 3306
```

```
    to_port     = 3306
    protocol    = "tcp"
    security_groups = [aws_security_group.app_sg.id]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "db-sg"
  }
}
```

Step 7: Launch EC2 Instances

Create instances for each tier: web, app, and database.

1. Add the following to instances.tf:

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.public.id
  security_groups = [aws_security_group.web_sg.name]
  key_name      = "your-key-pair"
  tags = {
    Name = "web-instance"
  }
}

resource "aws_instance" "app" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.app.id
  security_groups = [aws_security_group.app_sg.name]
  key_name      = "your-key-pair"
  tags = {
    Name = "app-instance"
  }
}

resource "aws_instance" "db" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.db.id
  security_groups = [aws_security_group.db_sg.name]
  key_name      = "your-key-pair"
  tags = {
    Name = "db-instance"
  }
}
```

Step 8: Configure Bastion Host

Secure access to private instances using a bastion host.

1. Add a bastion instance to instances.tf:

```
resource "aws_instance" "bastion" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.public.id
  key_name      = "your-key-pair"
  tags = {
    Name = "bastion-host"
  }
}
```

Step 9: Apply Terraform Configuration

1. Validate the configuration:

```
terraform validate
```

2. Preview the changes:

```
terraform plan
```

3. Apply the configuration:

```
terraform apply
```

4. Confirm the deployment when prompted.

Outcome

- **Infrastructure Components**:
    - A VPC with public and private subnets.

- A secure web, app, and database tier.
- A bastion host for secure access to private instances.

- **Access**:
  - Web tier accessible via the public subnet.
  - Secure communication between tiers using security groups.
  - SSH access to private instances through the bastion host.

Project 3: Automate Kubernetes Cluster Deployment on AWS Using Terraform

This project focuses on automating the deployment of a Kubernetes cluster on AWS using Amazon Elastic Kubernetes Service (EKS) with Terraform. It sets up a managed Kubernetes control plane, worker nodes, and networking infrastructure to host containerized applications.

Implementation Steps

Step 1: Set Up Terraform

1. **Install Terraform**: Ensure Terraform is installed on your system.
2. **Create a Project Directory**: Create a directory, e.g., terraform-eks-cluster, and navigate into it.

Step 2: Configure the AWS Provider

1. Create a file named main.tf and configure the AWS provider:

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

2. Initialize the project:

```
terraform init
```

Step 3: Create a VPC

Set up a Virtual Private Cloud (VPC) with public and private subnets for the EKS cluster.

1. Create a file named vpc.tf and define the VPC:

```
resource "aws_vpc" "eks_vpc" {
  cidr_block = "10.0.0.0/16"
  enable_dns_support = true
  enable_dns_hostnames = true
  tags = {
    Name = "eks-vpc"
  }
}
```

2. Add subnets for the VPC:

```
resource "aws_subnet" "public_subnet" {
  vpc_id      = aws_vpc.eks_vpc.id
  cidr_block = "10.0.1.0/24"
  map_public_ip_on_launch = true
  availability_zone = "us-east-1a"
  tags = {
    Name = "eks-public-subnet"
  }
}

resource "aws_subnet" "private_subnet" {
  vpc_id      = aws_vpc.eks_vpc.id
  cidr_block = "10.0.2.0/24"
  availability_zone = "us-east-1b"
  tags = {
    Name = "eks-private-subnet"
  }
}
```

3. Add an internet gateway and a route table:

```
resource "aws_internet_gateway" "eks_igw" {
  vpc_id = aws_vpc.eks_vpc.id
  tags = {
    Name = "eks-igw"
  }
}

resource "aws_route_table" "eks_public_route_table" {
  vpc_id = aws_vpc.eks_vpc.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.eks_igw.id
  }
}

resource "aws_route_table_association" "public_association" {
  subnet_id      = aws_subnet.public_subnet.id
  route_table_id = aws_route_table.eks_public_route_table.id
}
```

Step 4: Create an EKS Cluster

1. Create a file named eks.tf and define the EKS cluster:

```
module "eks" {
  source          = "terraform-aws-modules/eks/aws"
  cluster_name    = "eks-cluster"
  cluster_version = "1.25"
  subnets      = [aws_subnet.public_subnet.id, aws_subnet.private_subnet.id]
  vpc_id       = aws_vpc.eks_vpc.id

  node_groups = {
    eks_nodes = {
      desired_capacity = 2
      max_capacity     = 3
      min_capacity     = 1
      instance_type    = "t3.medium"
    }
  }

  tags = {
    Name = "eks-cluster"
  }
}
```

2. This uses a Terraform EKS module to simplify the deployment.

Step 5: Set Up IAM Roles and Policies

Define IAM roles and policies required for EKS.

1. Add IAM roles in iam.tf:

```
resource "aws_iam_role" "eks_role" {
  name = "eks-cluster-role"
  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "eks.amazonaws.com"
        }
      }]
    })

  tags = {
    Name = "eks-role"
  }
}

resource "aws_iam_role_policy_attachment" "eks_policy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"
  role       = aws_iam_role.eks_role.name
}
```

Step 6: Apply the Terraform Configuration

1. Validate the configuration:

```
terraform validate
```

2. Preview the infrastructure changes:

```
terraform plan
```

3. Apply the changes:

```
terraform apply
```

4. Confirm the deployment when prompted.

Step 7: Configure kubectl

1. Update your kubeconfig file to connect to the EKS cluster:

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

2. Verify the cluster connection:

```
kubectl get nodes
```

Step 8: Deploy a Sample Application

1. Create a deployment YAML file, e.g., app-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
spec:
 replicas: 2
 selector:
  matchLabels:
   app: nginx
 template:
  metadata:
   labels:
    app: nginx
  spec:
   containers:
   - name: nginx
    image: nginx:latest
```

```
        ports:
        - containerPort: 80
```

2. Apply the deployment:

```
kubectl apply -f app-deployment.yaml
```

3. Expose the application using a service:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: LoadBalancer
```

4. Apply the service:

```
kubectl apply -f service.yaml
```

5. Access the application via the Load Balancer URL.

Outcome

- **Infrastructure Components**:
  - A VPC with public and private subnets.
  - An EKS cluster with a managed control plane and worker nodes.
  - Networking and security configurations.

- A sample application deployed on Kubernetes.
- **Access**:
  - The sample application is accessible through the Load Balancer's DNS name.

Project 4: Automate the Deployment of a Complete CI/CD Pipeline on AWS Using Terraform

This project focuses on building a fully automated CI/CD pipeline on AWS using Terraform. The pipeline integrates AWS CodePipeline, CodeBuild, CodeCommit, and S3 for hosting and deploying a static website.

Implementation Steps

Step 1: Set Up Terraform

1. Install Terraform and create a new project directory, e.g., terraform-cicd-pipeline.
2. Configure the AWS provider in main.tf:

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

Step 2: Create an S3 Bucket for Hosting

1. Add the S3 bucket resource to s3.tf:

```
resource "aws_s3_bucket" "website_bucket" {
  bucket = "cicd-website-bucket"
  acl    = "public-read"

  website {
    index_document = "index.html"
  }

  tags = {
    Name = "CICD Website Bucket"
```

```
 }
}
```

2. Add a policy to make the bucket content publicly accessible:

```
resource "aws_s3_bucket_policy" "website_bucket_policy" {
 bucket = aws_s3_bucket.website_bucket.id

 policy = jsonencode({
  Version = "2012-10-17"
  Statement = [{
   Effect = "Allow"
   Principal = "*"
   Action = "s3:GetObject"
   Resource = "${aws_s3_bucket.website_bucket.arn}/*"
  }]
 })
}
```

Step 3: Create a CodeCommit Repository

1. Define the CodeCommit repository in codecommit.tf:

```
resource "aws_codecommit_repository" "source_repo" {
 repository_name = "cicd-demo-repo"
 description     = "CodeCommit repository for CI/CD demo"

 tags = {
  Name = "CI/CD Demo Repo"
 }
}
```

2. Initialize a local Git repository and push code to the CodeCommit repository:

```
git init
git remote add origin https://git-codecommit.us-east-1.amazonaws.com/v1/repos/cicd-demo-repo
git add .
git commit -m "Initial commit"
git push -u origin main
```

Step 4: Create a CodeBuild Project

1. Add the CodeBuild resource to codebuild.tf:

```
resource "aws_codebuild_project" "build_project" {
  name         = "cicd-build-project"
  service_role  = aws_iam_role.codebuild_role.arn
  source {
    type          = "CODECOMMIT"
    location       = aws_codecommit_repository.source_repo.clone_url_http
  }
  artifacts {
    type = "S3"
    location = aws_s3_bucket.website_bucket.id
    packaging = "ZIP"
  }
  environment {
    compute_type                = "BUILD_GENERAL1_SMALL"
    image                   = "aws/codebuild/standard:5.0"
    type                    = "LINUX_CONTAINER"
    environment_variables = [
      {
        name  = "S3_BUCKET"
        value = aws_s3_bucket.website_bucket.bucket
      }
    ]
  }

  tags = {
    Name = "Build Project"
  }
}
```

Step 5: Set Up IAM Roles

1. Create IAM roles in iam.tf for CodeBuild and CodePipeline:

```
resource "aws_iam_role" "codebuild_role" {
  name = "codebuild-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect = "Allow"
      Principal = {
        Service = "codebuild.amazonaws.com"
      }
      Action = "sts:AssumeRole"
    }]
  })

  tags = {
    Name = "CodeBuild Role"
  }
}

resource "aws_iam_role_policy_attachment" "codebuild_policy" {
  role       = aws_iam_role.codebuild_role.name
  policy_arn = "arn:aws:iam::aws:policy/AWSCodeBuildDeveloperAccess"
}

resource "aws_iam_role" "codepipeline_role" {
  name = "codepipeline-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect = "Allow"
      Principal = {
        Service = "codepipeline.amazonaws.com"
      }
      Action = "sts:AssumeRole"
    }]
  })

  tags = {
    Name = "CodePipeline Role"
  }
}

resource "aws_iam_role_policy_attachment" "codepipeline_policy" {
  role       = aws_iam_role.codepipeline_role.name
  policy_arn = "arn:aws:iam::aws:policy/AWSCodePipelineFullAccess"
}
```

Step 6: Configure CodePipeline

1. Add CodePipeline in codepipeline.tf:

```
resource "aws_codepipeline" "pipeline" {
  name     = "cicd-pipeline"
  role_arn = aws_iam_role.codepipeline_role.arn

  artifact_store {
    location = aws_s3_bucket.website_bucket.id
    type     = "S3"
  }

  stage {
    name = "Source"
    action {
      name           = "Source"
      category        = "Source"
      owner          = "AWS"
      provider        = "CodeCommit"
      version         = "1"
      output_artifacts = ["source_output"]
      configuration = {
        RepositoryName =
aws_codecommit_repository.source_repo.repository_name
        BranchName     = "main"
      }
    }
  }

  stage {
    name = "Build"
    action {
      name           = "Build"
      category        = "Build"
      owner          = "AWS"
      provider        = "CodeBuild"
      version         = "1"
      input_artifacts  = ["source_output"]
      output_artifacts = ["build_output"]
      configuration = {
        ProjectName = aws_codebuild_project.build_project.name
      }
    }
  }

  stage {
    name = "Deploy"
    action {
      name            = "Deploy"
      category        = "Deploy"
      owner           = "AWS"
      provider        = "S3"
      version         = "1"
      input_artifacts  = ["build_output"]
      configuration = {
        BucketName = aws_s3_bucket.website_bucket.bucket
        Extract    = "true"
      }
    }
  }

  tags = {
    Name = "CI/CD Pipeline"
  }
}
```

Step 7: Apply the Terraform Configuration

1. Validate the configuration:

```
terraform validate
```

2. Preview the infrastructure changes:

```
terraform plan
```

3. Apply the configuration:

```
terraform apply
```

4. Confirm the deployment when prompted.

Step 8: Test the CI/CD Pipeline

1. Push code changes to the CodeCommit repository to trigger the pipeline.
2. Verify the pipeline execution in the AWS Management Console.
3. Access the hosted website via the S3 bucket's public endpoint.

Outcome

- **Infrastructure Components**:
  - An S3 bucket for hosting a static website.
  - A CodeCommit repository for source code.
  - A CodeBuild project for building the application.
  - A CodePipeline to automate CI/CD.
- **Access**:
  - The static website is deployed to the S3 bucket and publicly accessible.

Project 5: Automate the Deployment of a Serverless Application Using AWS Lambda and Terraform

This project focuses on deploying a serverless application using AWS Lambda, API Gateway, and DynamoDB. Terraform automates the setup, including creating a Lambda function, configuring API Gateway to expose the function, and integrating DynamoDB as the database layer.

Implementation Steps

Step 1: Set Up Terraform

1. Install Terraform on your system.
2. Create a directory, e.g., terraform-serverless-app, for the project.
3. Initialize Terraform by creating a main.tf file and adding the AWS provider:

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

4. Run:

```
terraform init
```

Step 2: Create an S3 Bucket for Lambda Deployment Package

1. In s3.tf, create an S3 bucket to store the Lambda deployment package:

```
resource "aws_s3_bucket" "lambda_bucket" {
  bucket = "serverless-app-lambda-bucket"
  acl    = "private"

  tags = {
    Name = "Lambda Deployment Bucket"
```

```
 }
}
```

Step 3: Create a DynamoDB Table

1. In dynamodb.tf, define the DynamoDB table to store application data:

```
resource "aws_dynamodb_table" "app_table" {
  name         = "serverless-app-table"
  billing_mode  = "PAY_PER_REQUEST"
  hash_key      = "id"
  attribute {
    name = "id"
    type = "S"
  }

  tags = {
    Name = "Serverless App Table"
  }
}
```

Step 4: Create a Lambda Function

1. Prepare the Lambda function code in a lambda/ directory. For example, save the following Python code in lambda/app.py:

```
import json
import boto3
import os

dynamodb = boto3.resource("dynamodb")
```

```
table_name = os.environ["DYNAMODB_TABLE"]
table = dynamodb.Table(table_name)


def lambda_handler(event, context):
    if event["httpMethod"] == "POST":
        body = json.loads(event["body"])
        item = {"id": body["id"], "data": body["data"]}
        table.put_item(Item=item)
        return {"statusCode": 200, "body": json.dumps({"message": "Item added successfully"})}


    elif event["httpMethod"] == "GET":
        items = table.scan()["Items"]
        return {"statusCode": 200, "body": json.dumps(items)}


    return {"statusCode": 400, "body": json.dumps({"message": "Unsupported method"})}
```

2. Zip the Lambda function code:

```
cd lambda
zip app.zip app.py
```

3. Upload the zip file to the S3 bucket and reference it in Terraform. In lambda.tf:

```
resource "aws_lambda_function" "app_lambda" {
  function_name = "serverless-app-function"
  s3_bucket     = aws_s3_bucket.lambda_bucket.bucket
  s3_key        = "app.zip"
  runtime       = "python3.8"
  handler       = "app.lambda_handler"
  role          = aws_iam_role.lambda_execution_role.arn

  environment {
    variables = {
      DYNAMODB_TABLE = aws_dynamodb_table.app_table.name
    }
  }

  tags = {
    Name = "Serverless Lambda Function"
  }
}
```

Step 5: Create IAM Roles for Lambda

1. Define IAM roles in iam.tf to grant Lambda permissions:

```
resource "aws_iam_role" "lambda_execution_role" {
  name = "lambda-execution-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect = "Allow"
      Principal = {
        Service = "lambda.amazonaws.com"
      }
      Action = "sts:AssumeRole"
    }]
  })

  tags = {
    Name = "Lambda Execution Role"
  }
}

resource "aws_iam_role_policy_attachment" "lambda_dynamodb_policy" {
  role       = aws_iam_role.lambda_execution_role.name
  policy_arn = "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess"
}

resource "aws_iam_role_policy_attachment" "lambda_logging_policy" {
  role       = aws_iam_role.lambda_execution_role.name
  policy_arn = "arn:aws:iam::aws:policy/service-
role/AWSLambdaBasicExecutionRole"
}
```

Step 6: Configure API Gateway

1. Create an API Gateway to expose the Lambda function via HTTP. Add this to api_gateway.tf:

```
resource "aws_api_gateway_rest_api" "app_api" {
  name        = "serverless-app-api"
  description = "API Gateway for the serverless app"
}
```

```
resource "aws_api_gateway_resource" "app_resource" {
  rest_api_id = aws_api_gateway_rest_api.app_api.id
  parent_id   = aws_api_gateway_rest_api.app_api.root_resource_id
  path_part   = "items"
}

resource "aws_api_gateway_method" "app_method" {
  rest_api_id   = aws_api_gateway_rest_api.app_api.id
  resource_id   = aws_api_gateway_resource.app_resource.id
  http_method   = "ANY"
  authorization = "NONE"
}

resource "aws_api_gateway_integration" "lambda_integration" {
  rest_api_id = aws_api_gateway_rest_api.app_api.id
  resource_id = aws_api_gateway_resource.app_resource.id
  http_method = aws_api_gateway_method.app_method.http_method
  type        = "AWS_PROXY"
  integration_http_method = "POST"
  uri         = aws_lambda_function.app_lambda.invoke_arn
}

resource "aws_lambda_permission" "api_gateway_permission" {
  statement_id  = "AllowAPIGatewayInvoke"
  action        = "lambda:InvokeFunction"
  function_name = aws_lambda_function.app_lambda.function_name
  principal     = "apigateway.amazonaws.com"
  source_arn    = "${aws_api_gateway_rest_api.app_api.execution_arn}/*/*"
}
```

Step 7: Apply the Terraform Configuration

1. Initialize Terraform:

```
terraform init
```

2. Validate the configuration:

```
terraform validate
```

3. Plan the deployment:

```
terraform plan
```

4. Deploy the infrastructure:

```
terraform apply
```

5. Confirm when prompted.

Step 8: Test the Application

1. Note the API Gateway URL from the output of terraform apply.
2. Use curl or a tool like Postman to test the API:
   - Add an item (POST request):

```
curl -X POST -H "Content-Type: application/json" -d '{"id": "1", "data": "Hello, world!"}' <API_URL>/items
```

   - Retrieve all items (GET request):

```
curl -X GET <API_URL>/items
```

Outcome

- **Infrastructure Components**:

    - A DynamoDB table for data storage.
    - A Lambda function to handle HTTP requests and interact with DynamoDB.
    - An API Gateway to expose the Lambda function as a RESTful API.
    - An S3 bucket to store the Lambda deployment package.
- **Access**:
    - The application is accessible via the API Gateway URL.

For more DevOps projects* -> *CLICK HERE

Conclusion

Terraform has redefined how we approach infrastructure management by enabling a declarative and automated way of provisioning cloud resources. Its ability to support multi-cloud environments, simplify complex setups, and maintain consistency across deployments makes it an indispensable tool for developers, DevOps engineers, and cloud architects.

In this guide, we explored five practical and impactful projects to master Terraform:

1. Deploying a high-availability web application.
2. Building a secure multi-tier architecture.
3. Automating Kubernetes cluster provisioning.
4. Creating a complete CI/CD pipeline.
5. Implementing a serverless application with AWS Lambda and DynamoDB.

These projects covered a wide range of use cases, showcasing how Terraform can be applied to automate infrastructure, enhance scalability, and simplify maintenance. Each project was designed to help you gain hands-on experience with real-world scenarios, providing you with the knowledge to confidently work on cloud-based infrastructures.

By following the step-by-step implementation of these projects, you've not only learned how to build different components but also gained insights into Terraform best practices, such as modular design, role-based access, and secure resource management. These skills are essential for scaling applications, reducing downtime, and ensuring that your infrastructure can adapt to changing business needs.

Whether you're starting your journey with Terraform or looking to refine your existing skills, these projects offer a solid foundation for mastering infrastructure as code. Terraform's flexibility and powerful capabilities make it a key player in the DevOps ecosystem, empowering organizations to move faster and more efficiently in today's cloud-first world.

As you continue to explore Terraform, remember that the possibilities are endless. You can build on these projects, customize them for your unique requirements, and expand your expertise to include more advanced topics like state management, CI/CD pipelines for Terraform itself, and integrations with third-party tools.

With Terraform in your toolkit, you're well-equipped to tackle the challenges of modern infrastructure management. Keep experimenting, learning, and building, and you'll soon become a pro at automating infrastructure with Terraform!