

Terraform AWS Project

CloudStacker: "Terraform automated Three-tier Architecture on AWS"



By

Aravind U R

“CloudStacker: Automated Three-tier architecture on AWS”

This project aims to design and implement an **AWS-based three-tier architecture** using **Terraform**. The architecture will be integrated with **S3 for artifact storage**, **S3 for remote backend state file storage**, and **DynamoDB for state locking** to ensure consistent deployment.

Additionally, we will store the Terraform code in a **GitHub repository** for version control and collaboration. The project will then transition to **Terraform Cloud CI/CD** by removing the S3 remote backend and DynamoDB state lock in Favor of Terraform Cloud's built-in capabilities.

Here, I am mainly focusing on Terraform features such as Modularity, Remote Backend, and State Lock.

1. Modularity in Terraform

Definition:

Modularity in Terraform refers to the practice of organizing infrastructure code into reusable, self-contained blocks called **modules**. A module is a container for multiple resources that are used together.

Key Features:

- **Encapsulation:** Groups related resources and configurations.
- **Reusability:** Allows code to be reused across different environments or projects.
- **Maintainability:** Simplifies complex configurations by breaking them into smaller, manageable pieces.
- **Versioning:** Modules can be versioned to ensure consistency and avoid breaking changes.

Structure of a Module:

A module typically consists of:

- **main.tf:** Contains the resource definitions.
- **variables.tf:** Defines input variables for the module.
- **outputs.tf:** Specifies outputs that other configurations can use.

Benefits:

- Simplifies the configuration for large infrastructures.
- Ensures consistency across multiple deployments.
- Improves team collaboration by enabling shared module libraries.

2. Remote Backend

Definition:

A **backend** in Terraform is responsible for storing the state file, which tracks the real-world infrastructure Terraform manages. A **remote backend** stores the state file in a shared location, such as AWS S3, Azure Blob Storage, or HashiCorp's Terraform Cloud.

Why Use a Remote Backend?

- Enables collaboration among team members by allowing shared access to the state file.
- Secures the state file with encryption and access controls.
- Ensures high availability and disaster recovery by storing the state in a reliable location.

Benefits:

- **Team Collaboration:** Multiple team members can work on the same infrastructure without overwriting changes.
- **State Recovery:** Remote backends provide automatic backup and recovery options.
- **Scalability:** Suitable for large teams and complex infrastructures.

Example: Remote Backend Configuration (AWS S3)

[Terraform Docs:](#)

3. State Locking (State Lock)

Definition:

State locking ensures that the Terraform state file is not modified by multiple users or processes simultaneously. It prevents race conditions and corruption of the state file during operations like terraform apply or terraform plan.

How It Works:

- When a Terraform operation is initiated, terraform attempts to lock the state file.
- If the state file is already locked by another process, Terraform will wait or fail, depending on the configuration.
- The lock is released when the operation is completed successfully or manually.

Common Scenarios:

1. **Single User Locking:** Prevents conflicts when a single user runs multiple Terraform commands simultaneously.
2. **Team Locking:** Ensures only one team member modifies the state file at any time.

Benefits:

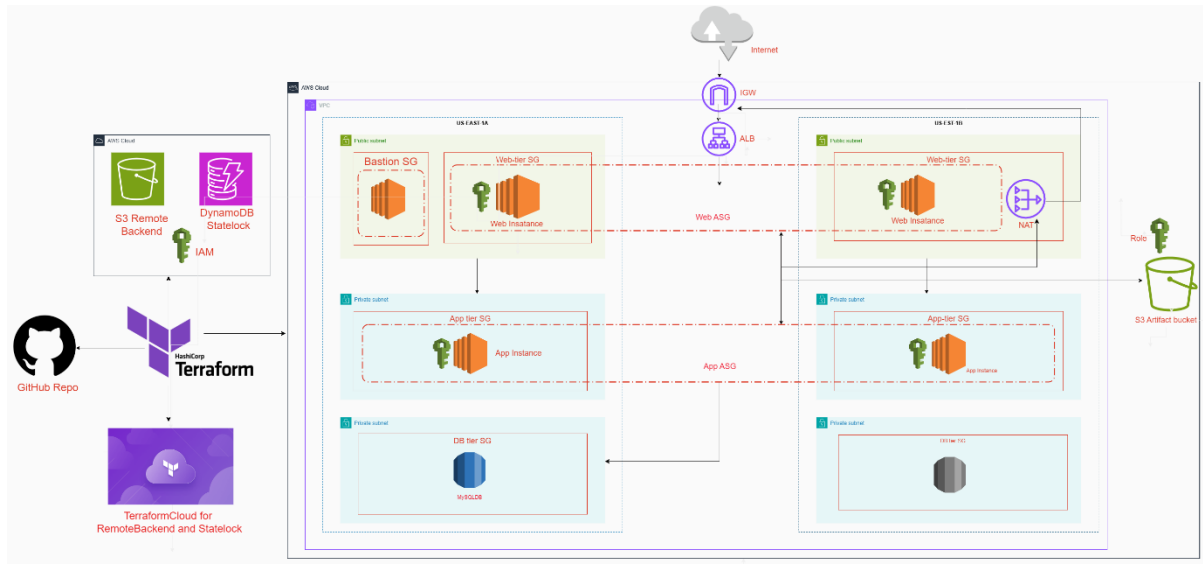
- Prevents infrastructure inconsistencies due to simultaneous changes.
- Protects the integrity of the state file in collaborative environments.
- Reduces the risk of deployment failures caused by state corruption.

In Shorts;

- **Modularity** improves code organization and promotes reusable infrastructure components.
- **Remote Backends** enhance collaboration and security in team environments.
- **State Locking** ensures safe and consistent management of infrastructure state, particularly in multi-user scenarios.

AWS Three-Tier Architecture

Diagram:



Overview:

The AWS Three-Tier Architecture is a common design pattern for building scalable, secure, and highly available web applications. It divides the architecture into three distinct layers: Presentation, Application, and Data. Each tier is responsible for a specific set of tasks, improving modularity, security, and scalability.

- **Three-Tier Architecture:**
 - **Web Tier:** Publicly accessible, handles incoming HTTP(S) traffic.
 - **Application Tier:** Processes business logic in a private subnet.
 - **Database Tier:** Stores application data securely in a private subnet.
- **Automation:**
 - Terraform manages the infrastructure provisioning.

- Remote backend using **S3** and **DynamoDB** is configured for state storage and locking.

Key Components:

Infrastructure Provisioning:

- Terraform Cloud.
- Remote backend for storing the Terraform state in S3 and locking with DynamoDB.
- Integrated with GitHub for storing Terraform code.

Networking:

- **VPC**: Hosts all resources within a single network.
- **Subnets**:
 - **Public Subnet**: For Bastion Host and Web Instances.
 - **Private Subnets**: For Application and Database tiers.
- **Internet Gateway (IGW)**: Allows public-facing resources to access the internet.
- **NAT Gateway**: Enables private subnets to access the internet securely.

Web Tier:

- **Auto Scaling Group (ASG)**: Manages multiple web instances for high availability.
- **Load Balancer (ALB)**: Distributes traffic to web instances in different availability zones (AZs).
- **Security Groups (SG)**:
 - **Web-Tier SG**: Allows HTTP/HTTPS traffic and communication with the Application tier.

Application Tier:

- **Auto Scaling Group (ASG)**: Manages multiple app instances for high availability.
- **App Instances**: Processes the logic and communicates with the database.
- **App-Tier SG**: Allows traffic only from the Web-Tier.

Database Tier:

- **RDS (MySQL):** Managed database service for secure and reliable storage.
- **DB-Tier SG:** Allows traffic only from the Application tier.

State Management:

- **S3 Bucket:** Stores Terraform state files.
- **DynamoDB Table:** Handles state locks to avoid conflicts during parallel runs.

Artifact Storage:

- An **S3 Artifact Bucket** is used for storing application files or build artifacts.

Bastion Host:

- Acts as a jump server for securely accessing instances in private subnets.
- Deployed in the public subnet with restricted access.

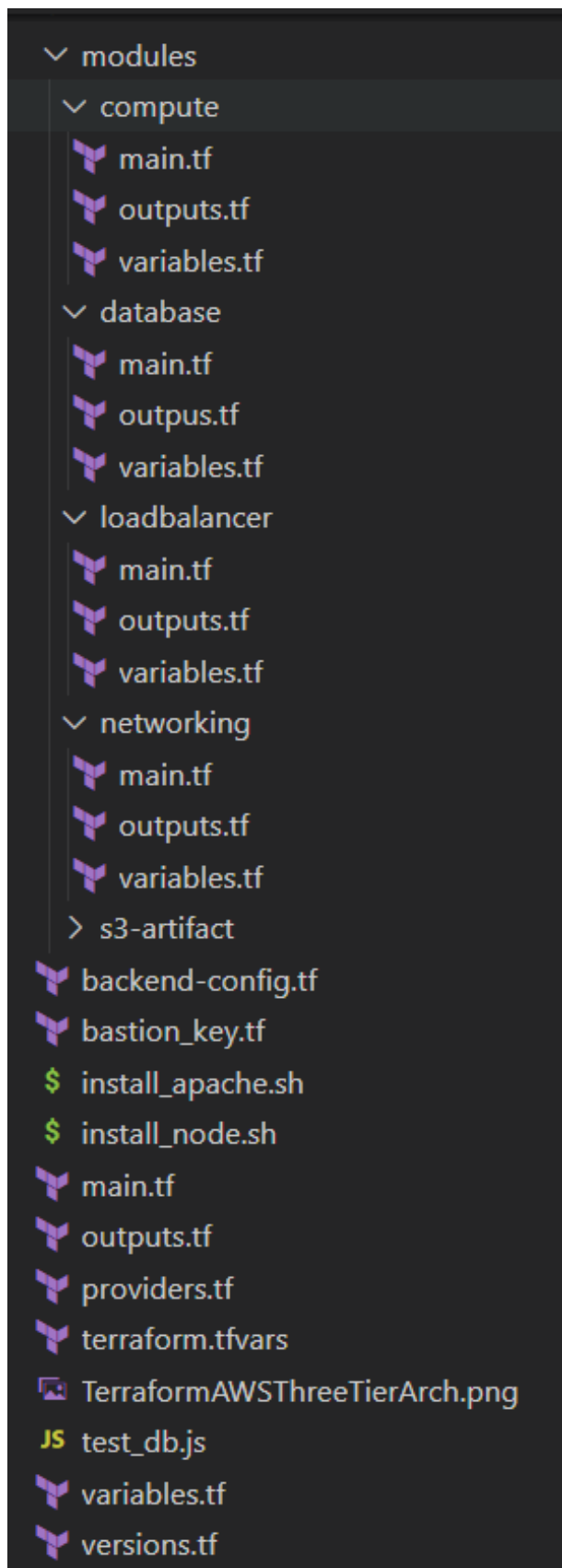
Terraform Code Section

Prerequisites

- An AWS account with IAM user access.
- Code Editor (I used VS Code for this deployment)
- AWS console — account with admin permissions, NOT a root account
- Terraform CLI
- Terraform installed on your local machine
<https://developer.hashicorp.com/terraform/downloads>.
- Terraform extension used in VS Code

The industry best practice for Terraform is to create separate files for each resource type. This makes the code more modular and easier to understand. It also makes it easier to reuse code and to test changes to the infrastructure.

I will briefly go over the filesystem I will be using for this project.



Providers:

providers.tf

```
provider "aws" {  
  region = var.aws_region  
}
```

provided the aws_region variable in root/variables.tf

versions.tf

```
terraform {  
  required_providers {  
    aws = {  
      source = "hashicorp/aws"  
      version = "~> 5.0"  
    }  
  }  
}
```

Modules:

Networking

This is the first module we will want to compose. It will contain the VPC, subnets, internet gateway, route tables, NAT gateways, security groups, and database subnet group.

main.tf : Defines the networking resources.

VPC Configuration

```
resource "aws_vpc" "three_tier_vpc" {  
  cidr_block = var.three_tier_vpc_cidr  
  enable_dns_hostnames = true  
  enable_dns_support = true  
  
  tags = {  
    Name = "three_tier_vpc"  
  }  
  
  lifecycle {  
    create_before_destroy = true  
  }  
}
```

internet gateway for project VPC

```
resource "aws_internet_gateway" "three_tier_igw" {  
  vpc_id = aws_vpc.three_tier_vpc.id  
  tags = {  
    Name = "three_tier_igw"  
  }  
}
```

```

lifecycle {
  create_before_destroy = true
}
}

```

Public subnets for web-tier

```

resource "aws_subnet" "three_tier_web_pubsub" {
  count = length(var.azs) #This will create a number of subnets equal to the number of Availability Zones in
var.azs. ie,2
  vpc_id = aws_vpc.three_tier_vpc.id
  cidr_block = var.subnets_cidrs[count.index] # CIDR for the web tier from list
  availability_zone = var.azs[count.index] # az's from the list
  map_public_ip_on_launch = true

  tags = {
    Name = "three_tier_web_pubsub_${substr(var.azs[count.index], -2, 2)}"
  }
}

```

Private subnets for app-tier

```

resource "aws_subnet" "three_tier_app_pvtsub" {
  count = length(var.azs) #This will create a number of subnets equal to the number of Availability Zones in
var.azs. ie,2
  vpc_id = aws_vpc.three_tier_vpc.id
  cidr_block = var.subnets_cidrs[count.index + 2] # CIDR for the web tier from list
  availability_zone = var.azs[count.index] # az's from the list
  map_public_ip_on_launch = false

  tags = {
    Name = "three_tier_app_pvtsub_${substr(var.azs[count.index], -2, 2)}"
  }
}

```

private subnets for database-tier

```

resource "aws_subnet" "three_tier_db_pvtsub" {
  count = length(var.azs)
  vpc_id = aws_vpc.three_tier_vpc.id
  cidr_block = var.subnets_cidrs[count.index + 4]
  availability_zone = var.azs[count.index]
  map_public_ip_on_launch = "false"

  tags = {
    Name = "three_tier_db_pvtsub_${substr(var.azs[count.index], -2, 2)}"
  }
}

```

elastic IP for NAT

```

resource "aws_eip" "eip_nat" {
  tags = {
    name = "eip_nat"
  }
}

```

NAT gateway for project vpc

```
resource "aws_nat_gateway" "three_tier_nat" {
  allocation_id = aws_eip.eip_nat.id
  subnet_id = aws_subnet.three_tier_web_pubsub[0].id
  tags = {
    Name = "three_tier_nat"
  }

  depends_on = [ aws_internet_gateway.three_tier_igw ]
}
```

public route table

```
resource "aws_route_table" "three_tier_public_rt" {
  vpc_id = aws_vpc.three_tier_vpc.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.three_tier_igw.id
  }

  tags = {
    Name = "three_tier_pub_rt"
  }
}
```

private route table

```
resource "aws_route_table" "three_tier_pvt_rt" {
  vpc_id = aws_vpc.three_tier_vpc.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_nat_gateway.three_tier_nat.id
  }

  tags = {
    Name = "three_tier_pvt_rt"
  }
}
```

route table association for web-tier subnets

```
resource "aws_route_table_association" "web_route_association" {
  count = length(var.azs)
  subnet_id = aws_subnet.three_tier_web_pubsub[count.index].id
  route_table_id = aws_route_table.three_tier_public_rt.id
}
```

route table association for app-tier subnets

```
resource "aws_route_table_association" "app_route_association" {
  count = length(var.azs)
  subnet_id = aws_subnet.three_tier_app_pvtsb[count.index].id
  route_table_id = aws_route_table.three_tier_pvt_rt.id
}
```

route table association for db-tier subnets

```
resource "aws_route_table_association" "db_route_association" {
  count = length(var.azs)
  subnet_id = aws_subnet.three_tier_db_pvtsub[count.index].id
  route_table_id = aws_route_table.three_tier_pvt_rt.id
}
```

#security groups

#Bastion security group

```
resource "aws_security_group" "three_tier_bastion_sg" {
  name = "bastion-sg"
  description = "allow ssh access from internet"
  vpc_id = aws_vpc.three_tier_vpc.id

  ingress {
    from_port = 22
    to_port = 22
    protocol = "tcp"
    cidr_blocks = [var.access_ip]
  }

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

  tags = {
    name = "bastion-sg"
  }
}
```

ALB SG

```
resource "aws_security_group" "three_tier_alb_sg" {
  description = "security group allows http and https from internet"
  name = "three-tier-alb-sg"
  vpc_id = aws_vpc.three_tier_vpc.id

  ingress {
    description = "allow http access from internet"
    from_port = 80
    to_port = 80
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    description = "allow all outbound rules"
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

```

}

tags = {
  name = "three_tier_alb_sg"
}

}

# frontend web tier sg

resource "aws_security_group" "three_tier_frontend_web_tier_sg" {
  description = "allow http inbound from loadbalancer sg and ssh inbound from bastion sg"
  name = "three-tier-frontend-web-sg"
  vpc_id = aws_vpc.three_tier_vpc.id

  ingress {
    description = "allow http inbound from loadbalancer security group"
    from_port = 80
    to_port = 80
    protocol = "tcp"
    security_groups = [aws_security_group.three_tier_alb_sg.id]
  }

  ingress {
    description = "Allow SSH from Bastion host"
    from_port = 22
    to_port = 22
    protocol = "tcp"
    security_groups = [aws_security_group.three_tier_bastion_sg.id] # Bastion SG as source
  }

  egress {
    description = "allow all outbound rules"
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    name = "three_tier_frontend_web_tier_sg"
  }
}

# backend app tier sg

resource "aws_security_group" "three_tier_backend_app_tier_sg" {
  description = "allow http inbound from frontend_web_sg and ssh inbound from bastion"
  name = "three-tier-backend-app-sg"
  vpc_id = aws_vpc.three_tier_vpc.id

  ingress {
    description = "allow all inbound from frontend_web_sg"
    from_port = 0
    to_port = 0
    protocol = "tcp"
  }
}

```

```

    security_groups = [aws_security_group.three_tier_frontend_web_tier_sg.id]
}

ingress {
    description = "allow ssh from bastion "
    from_port = 22
    to_port = 22
    protocol = "tcp"
    security_groups = [aws_security_group.three_tier_bastion_sg.id]
}

egress {
    description = "allow all outbound rules"
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
}

tags = {
    name = "three_tier_backend_app_tier_sg"
}
}

# backend(db tier) db sg

resource "aws_security_group" "three_tier_backend_db_tier_sg" {
    description = "allow MySQL port inbound from three_tier_backend_app_tier_sg"
    name = "three-tier-backend-db-tier-sg"
    vpc_id = aws_vpc.three_tier_vpc.id

    ingress {
        description = "allow MySQL port three_tier_backend_app_tier_sg"
        from_port = 3306
        to_port = 3306
        protocol = "tcp"
        security_groups = [aws_security_group.three_tier_backend_app_tier_sg.id]
    }

    egress {
        description = "allow all outbound rules"
        from_port = 0
        to_port = 0
        protocol = "-1"
        cidr_blocks = ["0.0.0.0/0"]
    }

    tags = {
        name = "three_tier_backend_db_tier_sg"
    }
}

#database subnet group

resource "aws_db_subnet_group" "three_tier_db_subnetgroup" {
    name = "three-tier-rds-subnetgroup"

```

```
subnet_ids = aws_subnet.three_tier_db_pvtsub[*].id

tags = {
  Name = "three_tier_rds_subnetgroup"
}
}
```

The *variables.tf* networking file and *outputs.tf* networking files are next below. Note that the variables I have listed must be present in the root *main.tf* file, and the outputs will be used throughout the rest of the configuration.

Variables.tf : Contains network configuration variables.

```
# --- networking/variables.tf ---

variable "azs" {
  type = list(string)
  description = "list of availability zones"
}

variable "three_tier_vpc_cidr" {
  type = string
  description = "cidr block of three_tier vpc"
}

variable "subnets_cidrs" {
  type = list(string)
  description = "List of CIDR blocks for subnets (web, app, and db tiers)"
}

variable "access_ip" {
  type = string
  description = "specific ip address only permit for ssh into bastion instance"
}
```

Outputs.tf : Specifies network-related outputs.

```
# --- networking/outputs.tf ---

output "three_tier_alb_sg_id" {
  value = aws_security_group.three_tier_alb_sg.id
  description = "id of alb sg"
}

output "three_tier_vpc_id" {
  value = aws_vpc.three_tier_vpc.id
  description = "id of three tier vpc"
}
```



```

output "three-tier-db-subnetgroup_name" {
  value = aws_db_subnet_group.three_tier_db_subnetgroup.name
  description = "name of the db subnet group"
}

output "three-tier-db-subnetgroup_id" {
  value = aws_db_subnet_group.three_tier_db_subnetgroup.id
  description = "id of the db subnet group"
}

output "three_tier_backend_db_tier_sg" {
  value = aws_security_group.three_tier_backend_db_tier_sg.id
  description = "id of the db security group"
}

output "three_tier_bastion_sg" {
  value = aws_security_group.three_tier_bastion_sg.id
  description = "id of the bastion security group"
}

output "three_tier_alb_sg" {
  value = aws_security_group.three_tier_alb_sg.id
  description = "id of the alb security group"
}

output "three_tier_frontend_web_tier_sg" {
  value = aws_security_group.three_tier_frontend_web_tier_sg.id
  description = "id of the web tier security group"
}

output "three_tier_backend_app_tier_sg" {
  value = aws_security_group.three_tier_backend_app_tier_sg.id
  description = "id of the app tier security group"
}

output "three_tier_web_pubsub" {
  value = aws_subnet.three_tier_web_pubsub.*.id
  description = "id's of the public subnets "
}

output "three_tier_app_pvsub" {
  value = aws_subnet.three_tier_app_pvsub.*.id
  description = "id's of the private subnets"
}

```

Compute

In this module, we will create launch templates and an auto scaling group for the corresponding Bastion, Web, and App tier instances. The Bastion and Web tiers will be in public subnets, while the App tier will be in private subnets. In the `bastion_key.tf` file, we have defined the SSH key pair using the `tls_provider`, and the key name is referenced in the instances for SSH access. The `install_apache.sh` script is used as user data for Apache installation on Web instances, and the `install_node.sh` script is used as user data for App tier instances.

main.tf : Defines the compute resources

```
# --- compute/main.tf ---
# launch template and auto scaling group for bastion

resource "aws_launch_template" "three_tier_bastion" {
  name           = "three-tier-bastion"
  image_id       = var.ami_value
  instance_type  = var.instance_type
  vpc_security_group_ids = [var.three_tier_bastion_sg]
  key_name       = var.key_name

  tags = {
    Name = "three_tier_bastion"
  }
}

resource "aws_autoscaling_group" "three_tier_bastion_asg" {
  name = "three-tier-bastion-ASG"
  vpc_zone_identifier = var.three_tier_web_pubsub
  max_size           = 1
  min_size           = 1
  desired_capacity    = 1

  launch_template {
    id = aws_launch_template.three_tier_bastion.id
    version = "$Latest"
  }

  tag {
    key      = "Name"
    value    = "three_tier_bastion"
    propagate_at_launch = true
  }
}
```

```
# launch template and auto scaling group for frontend web-app on web tier
```

```

resource "aws_launch_template" "three_tier_frontend_web_lt" {
  name          = "three-tier-frontend-web-lt"
  image_id      = var.ami_value
  instance_type = var.instance_type
  vpc_security_group_ids = [var.three_tier_frontend_web_tier_sg]
  key_name      = var.key_name
  iam_instance_profile {
    name = "s3_artifact_instance_profile"
  }
  user_data = filebase64("install_apache.sh")

  tags = {
    Name = "three_tier_frontend_web"
  }
}

resource "aws_autoscaling_group" "three_tier_frontend_web_asg" {

  name = "three-tier-frontend-web-asg"
  vpc_zone_identifier = var.three_tier_web_pubsub
  max_size           = 3
  min_size           = 2
  desired_capacity    = 2

  launch_template {
    id      = aws_launch_template.three_tier_frontend_web_lt.id
    version = "$Latest"
  }

  target_group_arns = [var.aws_alb_target_group_arn]

  tag {
    key          = "Name"
    value        = "three_tier_frontend_web"
    propagate_at_launch = true
  }

  health_check_type = "EC2"
}

resource "aws_autoscaling_attachment" "asg_attach" {
  autoscaling_group_name = aws_autoscaling_group.three_tier_frontend_web_asg.id
  lb_target_group_arn    = var.aws_alb_target_group_arn
}

```

```
# launch template and auto scaling group for backend app on app tier
```

```

resource "aws_launch_template" "three_tier_backend_app_lt" {
  name_prefix      = "three-tier-backend-app-lt"
  image_id         = var.ami_value
  instance_type    = var.instance_type
  vpc_security_group_ids = [var.three_tier_backend_app_tier_sg]
  key_name         = var.key_name
  iam_instance_profile {
    name = "s3_artifact_instance_profile"
  }
  user_data = filebase64("install_node.sh")

  tags = {
    Name = "three_tier_backend_app"
  }
}

resource "aws_autoscaling_group" "three_tier_backend_app_asg" {
  name = "three-tier-backend-app-asg"
  vpc_zone_identifier = var.three_tier_app_pvtsb
  max_size           = 3
  min_size           = 2
  desired_capacity   = 2

  launch_template {
    id = aws_launch_template.three_tier_backend_app_lt.id
    version = "$Latest"
  }

  tag {
    key      = "Name"
    value    = "three_tier_backend_app"
    propagate_at_launch = true
  }
}

```

variables.tf: Contains variable definitions for compute configurations.

```

#----compute/variables.tf-----

variable "key_name" {
  type = string
  description = "aws key pair name"
}

variable "ami_value" {
  type = string
  description = "ami id used for autoscaling groups"
}

```

```

variable "instance_type" {
  type = string
  description = "instance type for autoscaling groups "
}

variable "three_tier_bastion_sg" {
  type = string
  description = "The ID of the security group for the bastion host"
}

variable "three_tier_frontend_web_tier_sg" {
  type = string
  description = "The ID of the security group for the web tier instances"
}

variable "three_tier_backend_app_tier_sg" {
  type = string
  description = "The ID of the security group for the web tier instances"
}

variable "three_tier_app_pvtsub" {
  type = list(string)
  description = "the ID's of the app tier private subnets in both az's"
}

variable "three_tier_web_pubsub" {
  type = list(string)
  description = "the ID's of the web tier public subnets in both az's"
}

variable "aws_alb_target_group_name" {
  type = string
  description = "the name of the load balancer target group"
}

variable "aws_alb_target_group_arn" {
  type = string
  description = "the ARN of the load balancer target group"
}

```

Outputs.tf : Specifies output values related to compute resources

```

# --- compute/outputs.tf ---

output "three_tier_frontend_web_asg" {
  value = aws_autoscaling_group.three_tier_frontend_web_asg.arn
  description = "The ARN of the frontend web Auto Scaling Group"
}

output "three_tier_backend_app_asg" {
  value = aws_autoscaling_group.three_tier_backend_app_asg.arn
}

```

```
description = "The ARN of the backend app Auto Scaling Group"
}
```

Bastion_key.tf

```
# keypair for bastion

# generate tls private key locally

resource "tls_private_key" "three_tier_bastion_key" {
  algorithm = "RSA"
  rsa_bits = 2048
}

# save the privatekey locally for ssh access

resource "local_file" "three_tier_bastion_key" {
  content = tls_private_key.three_tier_bastion_key.private_key_pem
  filename =
"D:/Aravind/MyCourses/Terraform/AWSTerraform_ThreeTierArch/three_tier_bastion_key.pem"
  file_permission = "0400"
}

# generate aws keypair using public key created

resource "aws_key_pair" "three_tier_bastion_key" {
  key_name = var.key_name
  public_key = tls_private_key.three_tier_bastion_key.public_key_openssh
}
```

Load balancer

For the *main.tf* file here, we will simply create the load balancer, target group, and listener. Note that the load balancer lies in the public subnet layer. I also made sure to specify that the load balancer depend on the autoscaling group it is associated with, so that health checks were not failed.

main.tf: Specifies the load balancer resources

```
# --- loadbalancer/main.tf ---
# Internet facing application load balancer

resource "aws_alb" "three_tier_alb" {
  name = "three-tier-alb"
  internal = false
  load_balancer_type = "application"
  security_groups = [var.three_tier_alb_sg]
  subnets = var.three_tier_web_pubsub
}
```

```

idle_timeout = 400

depends_on = [
  var.three_tier_frontend_web_asg
]
}

# application load balancer target groups

# alb tg for http

resource "aws_alb_target_group" "three_tier_alb_tg_http" {
  name = "three-tier-alb-tg-http"
  port = var.tg_port
  protocol = var.tg_protocol
  vpc_id = var.three_tier_vpc_id

  lifecycle {
    ignore_changes = [ name ]
    create_before_destroy = true
  }

  tags = {
    Name = "three_tier_alb_tg_http"
  }
}

# application load balancer listener

# alb listener for http

resource "aws_alb_listener" "three_tier_alp_listener_http" {
  load_balancer_arn = aws_alb.three_tier_alb.arn
  port = var.listener_port
  protocol = var.listener_protocol

  default_action {
    type = "forward"
    target_group_arn = aws_alb_target_group.three_tier_alb_tg_http.arn
  }

  tags = {
    Name = "three_tier_alp_listener_http"
  }
}

```

variables.tf: Contains variable definitions for the load balancer setup

```
# --- loadbalancing/variables.tf ---
```

```

variable "three_tier_alb_sg" {
  type = string
  description = "The ID of the security group for the alb"
}
variable "three_tier_web_pubsub" {
  type = list(string)
  description = "the ID's of the web tier public subnets in both az's"
}
variable "three_tier_frontend_web_asg" {
  type = string
  description = "the ARN of the frontend web Auto Scaling Group"
}
variable "tg_port" {
  type = number
  description = "tg group port"
}

variable "tg_protocol" {
  type = string
  description = "target group protocol"
}

variable "three_tier_vpc_id" {
  type = string
  description = "id of the three tier vpc"
}

variable "listener_protocol" {
  type = string
  description = "alb listener protocol"
}

variable "listener_port" {
  type = number
  description = "alb listener port"
}

```

outputs.tf: Provides output data related to the load balancer.

```

# --- loadbalancing/outputs.tf ---

output "three_tier_alb_endpoint" {
  value = aws_alb.three_tier_alb.dns_name
  description = "dns of alb endpoint"
}

output "aws_alb_target_group_name" {
  value = aws_alb_target_group.three_tier_alb_tg_http.name
  description = "name of alb tg"
}

```



```
output "aws_alb_target_group_arn" {
  value = aws_alb_target_group.three_tier_alb_tg_http.arn
  description = "ARN of alb tg"
}
```

Database

Onto the last module. For the *main.tf* file here, we have a block that builds the MySQL database.

main.tf : Defines the database resources.

```
# --- database/main.tf ---
# three tier mysql db

resource "aws_db_instance" "three_tier_mysql_db" {
  allocated_storage = 10
  engine = var.db_engine
  engine_version = var.db_engine_version
  instance_class = var.db_instance_class
  username = var.db_username
  password = var.db_password
  db_subnet_group_name = var.three-tier-db-subnetgroup_name
  vpc_security_group_ids = [var.three_tier_backend_db_tier_sg_id]
  identifier = var.db_identifier
  skip_final_snapshot = "true"
  tags = {
    Name = "three_tier_mysql_db"
  }
}
```

variables.tf: Contains variable definitions for the database.

```
# --- database/variables.tf ---

variable "db_username" {
  type = string
  description = "three tier mysql db username"
}

variable "db_password" {
  type = string
  description = "three tier mysql db password"
}

variable "db_engine" {
  type = string
  description = "three tier mysql db engine type"
}
```

```

}

variable "db_engine_version" {
  type = string
  description = "three tier mysql db engine version"
}

variable "db_instance_class" {
  type = string
  description = "three tier mysql db instance class(type)"
}

variable "db_identifier" {
  type = string
  description = "three tier mysql db identifier"
}

variable "three-tier-db-subnetgroup_name" {
  type = string
  description = "name of the db subnet group"
}

variable "three_tier_backend_db_tier_sg_id" {
  type = string
  description = "id of security group assigned to backend db"
}

```

outputs.tf: Specifies output values for database-related data.

```

# --- database/outputs.tf ---

output "three_tier_mysql_db_endpoint" {
  value = aws_db_instance.three_tier_mysql_db.endpoint
  description = "endpoint url for mysql db"
}

```

S3-artifact

This Terraform module creates resources to manage access to an S3 bucket used for artifact storage and assigns the necessary permissions to Web and App tier EC2 instances for interacting with the bucket.

The **force_destroy** option ensures that the bucket and its contents can be deleted without manual intervention.

To prevent unauthorized public access to the bucket by blocking public ACLs, public policies, and restricting the bucket from being publicly accessed.

The module defines an IAM policy (`aws_iam_policy.s3_artifact_policy`) that grants the necessary permissions (`GetObject`, `PutObject`, `ListBucket`, `DeleteObject`) to access the artifacts stored in the S3 bucket.

An IAM role (`aws_iam_role.s3_artifact_role`) is created with an assume role policy allowing EC2 instances to assume this role to access the S3 bucket.

The IAM role is associated with the previously defined IAM policy (`aws_iam_role_policy_attachment.s3_artifact_role_policy_attachment`), ensuring that EC2 instances can access the S3 bucket using the role's permissions.

An IAM instance profile (`aws_iam_instance_profile.s3_artifact_instance_profile`) is created to be attached to EC2 instances, enabling them to assume the IAM role and access the S3 bucket.

main.tf : configurations for storing artifacts in an S3 bucket.

```
# --- S3_artifact/main.tf ---

# Define s3 bucket for artifact storage

resource "aws_s3_bucket" "s3_artifact_bucket" {
  bucket = var.s3_artifact_bucket
  force_destroy = true
}

# block public access

resource "aws_s3_bucket_public_access_block" "s3_artifact_public_block" {
  bucket = var.s3_artifact_bucket

  block_public_acls      = true
  block_public_policy    = true
  ignore_public_acls     = true
  restrict_public_buckets = true
}
```

```
# IAM policy for s3 artifact access

resource "aws_iam_policy" "s3_artifact_policy" {
  name = "s3-artifact-access-policy"
  path = "/"
  description = "allow access to s3"

  policy = jsonencode({
```

```

Version = "2012-10-17"
Statement = [
  {
    "Sid" : "S3ArtifactAllowAccess",
    Action = [
      "s3:GetObject",
      "s3:PutObject",
      "s3:ListBucket",
      "s3:DeleteObject"
    ]
    Effect = "Allow"
    Resource = [
      "arn:aws:s3:::*/*",
      "arn:aws:s3:::var.s3_artifact_bucket",
      "arn:aws:s3:::var.s3_artifact_bucket/*"
    ]
  },
]
})
}

```

IAM role for s3 artifact access

```

resource "aws_iam_role" "s3_artifact_role" {
  name = "s3_artifact_role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Sid   = ""
        Principal = {
          Service = "ec2.amazonaws.com"
        }
      },
    ]
  })
}

```

IAM role policy attachment

```

resource "aws_iam_role_policy_attachment" "s3_artifact_role_policy_attachment" {
  role      = aws_iam_role.s3_artifact_role.name
  policy_arn = aws_iam_policy.s3_artifact_policy.arn
}

```

IAM instance profile

```
resource "aws_iam_instance_profile" "s3_artifact_instance_profile" {
  name = "s3_artifact_instance_profile"
  role = aws_iam_role.s3_artifact_role.name
}
```

Variables.tf

```
variable "s3_artifact_bucket" {
  description = "Name of the S3 bucket for artifact storage"
}
```

Outputs.tf

```
output "s3_artifact_bucket_name" {
  value = aws_s3_bucket.s3_artifact_bucket.id
  description = "name of the s3 artifact bucket"
}

output "s3_artifact_role_arn" {
  value = aws_iam_role.s3_artifact_role.arn
  description = "arn of the role for accessing s3 artifact"
}

output "aws_iam_instance_profile_name" {
  value = aws_iam_instance_profile.s3_artifact_instance_profile.name
  description = "The name of the IAM instance profile associated with the S3 artifact role."
}
```

Finally, come at the root, we will add the ;

Here are the scripts I used to install Apache and Node.js. The Apache script will allow you to test the internet facing load balancer;

```
#----install_apache.sh-----
#!/bin/bash
yum update -y
yum install httpd -y
systemctl start httpd
systemctl enable httpd
echo "hello world from $(hostname -f)" | tee /var/www/html/index.html
```

```
#----install_node.sh-----
#!/bin/bash
yum update -y
yum -y install curl
yum install -y gcc-c++ make
curl -sL https://rpm.nodesource.com/setup_16.x | bash -
```

```
yum install -y nodejs
```

main.tf, ***variables.tf*** file, ***outputs.tf*** file, and the ***tfvars*** file:

root/main.tf

Here the main.tf file in this Terraform configuration defines the setup of our multi-tier infrastructure using modules of the architecture, including networking, compute resources, load balancers, databases, and artifact storage. It also specifies the backend for managing Terraform state remotely using Amazon S3.

```
# define s3 remote backend
terraform {
  backend "s3" {
    bucket = "threetierremotebackend"
    dynamodb_table = "remotebackend_statelock_dynamodb"
    key = "threetierremotebackend/terraform.tfstate"
    encrypt = true
    region = "us-east-1"
  }
}
```

```
/*The networking module defines the VPC and its associated subnets, security groups, and availability zones.
Variables such as the VPC CIDR block, access IP, and subnet CIDRs are passed into this module to configure the
networking setup*/
```

```
module "networking" {
  source = "./modules/networking"
  three_tier_vpc_cidr = var.three_tier_vpc_cidr
  access_ip = var.access_ip
  subnets_cidrs = var.subnets_cidrs
  azs = var.azs
}
```

```
/*The compute module creates EC2 instances for different tiers (e.g., Bastion, Web, and App tiers).It uses
security groups from the networking module and references other resources like the AMI value, instance type,
and ALB target group to configure EC2 instances.*/
```

```
module "compute" {
  source = "./modules/compute"
  key_name = var.key_name
  ami_value = var.ami_value
  instance_type = var.instance_type
  three_tier_bastion_sg = module.networking.three_tier_bastion_sg
  three_tier_frontend_web_tier_sg = module.networking.three_tier_frontend_web_tier_sg
  three_tier_backend_app_tier_sg = module.networking.three_tier_backend_app_tier_sg
}
```

```

three_tier_app_pvtsb = module.networking.three_tier_app_pvtsb
three_tier_web_pubsub = module.networking.three_tier_web_pubsub
aws_alb_target_group_arn = module.loadbalancer.aws_alb_target_group_arn
aws_alb_target_group_name = module.loadbalancer.aws_alb_target_group_name
}

```

/*The loadbalancer module sets up an Application Load Balancer (ALB) for distributing traffic across the web tier.It configures the security groups, target groups, and listeners, including port and protocol settings (HTTP on port 80).

The output block defines an output variable (three_tier_alb_endpoint) to expose the URL of the ALB for external access. */

```

module "loadbalancer" {
  source = "../modules/loadbalancer"
  three_tier_alb_sg = module.networking.three_tier_alb_sg
  three_tier_web_pubsub = module.networking.three_tier_web_pubsub
  three_tier_frontend_web_asg = module.compute.three_tier_frontend_web_asg
  tg_port = 80
  tg_protocol = "HTTP"
  three_tier_vpc_id = module.networking.three_tier_vpc_id
  listener_protocol = "HTTP"
  listener_port = 80
}

output "three_tier_alb_endpoint" {
  value = module.loadbalancer.three_tier_alb_endpoint
}

```

/*The database module provisions a managed database (e.g., RDS) using the specified database engine, version, and instance class.It uses subnets from the networking module and configures security groups for secure database access.*/

```

module "database" {
  source = "../modules/database"
  db_engine = var.db_engine
  db_engine_version = var.db_engine_version
  db_instance_class = var.db_instance_class
  db_username = var.db_username
  db_identifier = var.db_identifier
  db_password = var.db_password
  three-tier-db-subnetgroup_name = module.networking.three-tier-db-subnetgroup_name
  three_tier_backend_db_tier_sg_id = module.networking.three_tier_backend_db_tier_sg
}

```

/*The s3-artifact module creates an S3 bucket for artifact storage. The bucket name is passed as a variable to the module.*/

```

module "s3-artifact" {
  source = "../modules/s3-artifact"
  s3_artifact_bucket = var.s3_artifact_bucket
}

```

root/variables.tf

```
variable "aws_region" {
  type = string
  description = "aws region of our architecture"
}

variable "azs" {
  type = list(string)
  description = "list of availability zones"
}

variable "three_tier_vpc_cidr" {
  type = string
  description = "cidr block of three_tier vpc"
}

variable "subnets_cidrs" {
  type = list(string)
  description = "List of CIDR blocks for subnets (web, app, and db tiers)"
}

variable "key_name" {
  type = string
  description = "aws key pair name"
}

variable "ami_value" {
  type = string
  description = "ami id used for autoscaling groups"
}

variable "instance_type" {
  type = string
  description = "instance type for autoscaling groups "
}

variable "db_username" {
  type = string
  description = "three tier mysql db username"
}

variable "db_password" {
  type = string
  description = "three tier mysql db password"
}

variable "db_engine" {
  type = string
  description = "three tier mysql db engine type"
}

variable "db_engine_version" {
```



```

    type = string
    description = "three tier mysql db engine version"
}

variable "db_instance_class" {
    type = string
    description = "three tier mysql db instance class(type)"
}

variable "db_identifier" {
    type = string
    description = "three tier mysql db identifier"
}

variable "access_ip" {
    type = string
    description = "sepcific ip address only permit for ssh into bastion instance"
}

variable "s3_artifact_bucket" {
    description = "Name of the S3 bucket for artifact storage"
}

```

root/outputs.tf

```

output "load_balancer_endpoint" {
    value = module.loadbalancer.three_tier_alb_endpoint
}

output "db_endpoint" {
    value = module.database.three_tier_mysql_db_endpoint
}

```

root/terraform.tfvars

```

# aws region for our architecture
aws_region = "us-east-1"

# availability zones
azs = ["us-east-1a" , "us-east-1b"]

#cidr block of vpc
three_tier_vpc_cidr = "10.0.0.0/16"

# list of CIDR blocks for subnets
subnets_cidrs = [
    "10.0.1.0/24", # Web subnet 1a
    "10.0.2.0/24", # Web subnet 1b
    "10.0.3.0/24", # App subnet 1a
    "10.0.4.0/24", # App subnet 1b
    "10.0.5.0/24", # DB subnet 1a
    "10.0.6.0/24" # DB subnet 1b
]

```

```

]

# aws key pair name
key_name = "three_tier_bastion_key"

# ami value
ami_value = "ami-0166fe664262f664c"

# instance type value
instance_type = "t2.micro"

# three tier mysql db username
db_username = "db_admin"

# three tier mysql db password
db_password = "Admin123"

# three tier mysql db engine type
db_engine = "mysql"

# three tier mysql db engine version
db_engine_version = "8.0"

# three tier mysql db instance class(type)
db_instance_class = "db.t3.micro"

# three tier mysql db identifier
db_identifier = "ThreeTierDB"

# access ip for bastion instance
access_ip = "0.0.0.0/0"

# name of the S3 bucket for artifact storage
s3_artifact_bucket = "threetiers3artifact"

```

Terraform Commands

Now that the infrastructure as code is set up, we can apply it to our AWS account. Initially, I have commented out the backend block in your Terraform configuration. This means Terraform used a local backend by default to store its state file (terraform.tfstate) in your working directory.

Then from the root directory, run a ***terraform init***, then ***terraform validate*** if you want to see the validity of your code, ***terraform plan*** to map out the resources you will create, and terraform apply to execute the plan!

After your plan, you will see the number of resources to be created, and the outputs you will be shown. applied your configuration with terraform apply, which provisioned your AWS resources. The state file was saved locally.

```
Apply complete! Resources: 51 added, 0 changed, 0 destroyed.
```

Outputs:

```
db_endpoint = "threetierdb.cdceca62s40w.us-east-1.rds.amazonaws.com:3306"  
load_balancer_endpoint = "three-tier-alb-835443737.us-east-1.elb.amazonaws.com"  
three tier alb endpoint = "three-tier-alb-835443737.us-east-1.elb.amazonaws.com"
```

Terraform Remote Backend Init

Next, we have to uncomment the backend block in your Terraform configuration. This defines a remote backend (an AWS S3 bucket) where Terraform stores the state file instead of keeping it locally.

Running terraform init after uncommenting the backend block reconfigured Terraform to use the specified remote backend. Terraform likely prompted you to migrate the existing local state to the remote backend.

Terraform State File: The state file tracks the resources created by Terraform. By default, it's stored locally unless you configure a remote backend.

Remote Backend: The remote backend provides a centralized and secure location for the state file. This is crucial for collaboration and ensuring state consistency when working in a team.

Check State File Migration

After uncommenting the backend block and running terraform init, Terraform should output like:

```
PS D:\Aravind\MyCourses\Terraform\AWSTerraform_ThreeTierArch> terraform init  
Initializing the backend...  
Do you want to copy existing state to the new backend?  
Pre-existing state was found while migrating the previous "local" backend to the  
newly configured "s3" backend. No existing state was found in the newly  
configured "s3" backend. Do you want to copy this state to the new "s3"  
backend? Enter "yes" to copy and "no" to start with an empty state.  
  
Enter a value: █
```

If you confirmed by “yes”, it would migrate the state from local to the remote backend.

```

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
Initializing modules...
Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Reusing previous version of hashicorp/tls from the dependency lock file
- Reusing previous version of hashicorp/local from the dependency lock file
- Using previously-installed hashicorp/local v2.5.2
- Using previously-installed hashicorp/aws v5.82.2
- Using previously-installed hashicorp/tls v4.0.6

Terraform has been successfully initialized!

```

How to verify:

- Check your local directory: The terraform.tfstate file should no longer exist locally.
- Check the remote backend (e.g., AWS S3): The terraform.tfstate file should now appear in the configured S3 bucket.

Run ***terraform plan*** ;

To ensure Terraform can access the state file from the remote backend and correctly detect the current infrastructure state. The output should indicate that no changes are needed if the resources are already provisioned.

We can verify the DynamoDB table partition key shown in AWS console as below;

Tables (1) [Info](#)

<input type="checkbox"/>	Name	Status	Partition key	Sort key	Index
<input type="checkbox"/>	remotebackend_statelock_dynamodb	Active	LockID (S)	-	

Testing the Architecture

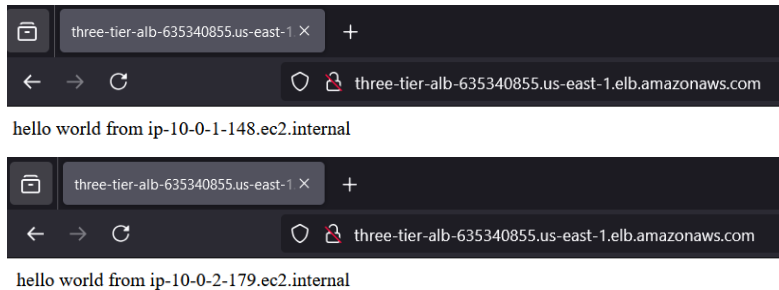
Steps to verify:

1. Connect to the bastion host:

- SSH into bastion ec2
- Create a file “three_tier_bastion_key.pem” and copy the public key details to the file.
- Edit the permission with “**chmod 0400 three_tier_bastion_key.pem**” command.

2. Check connection b/w ALB and WebTier

If we copy the load balancer endpoint we got from our Terraform output, and place it in the search bar, we will see the message we specified in our script for the Apache webserver.



3. Verify connection b/w App Tier and DB Tier

- SSH into App instance from bastions host.
- Verify the node version by ***node -v***

```
[ec2-user@ip-10-0-3-126 ~]$ node -v
v16.20.2
```

- Install mysql command line by ***sudo yum install -y mysql***
- Install mysql2 by ***npm install mysql2***

MySQL: The MySQL server (which is installed on the EC2 instance or Amazon RDS) is responsible for hosting the database. This allows you to store and query data.

mysql2: This is a **Node.js package** that provides an interface for **Node.js applications** to connect to and query a MySQL (or MariaDB) database.

It allows your JavaScript code to interact with the database.

*When you use Node.js to connect to MySQL (whether it's hosted locally on your EC2 instance or on Amazon RDS), you need a way to send queries to MySQL, and that's where the **mysql2 package** comes in.*

- Connect to MySQL DB from App tier EC2 Instance

```
[ec2-user@ip-10-0-3-94 ~]$ mysql -h threetierdb.cdceca62s40w.us-east-1.rds.amazonaws.com -u db_admin -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 35
Server version: 8.0.39 Source distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]>
```

- Create a database “threetierdb”

```
[ec2-user@ip-10-0-3-94 ~]$ mysql -h threetierdb.cdceca62s40w.us-east-1.rds.amazonaws.com -u db_admin -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 50
Server version: 8.0.39 Source distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]> CREATE DATABASE threetierdb;
Query OK, 1 row affected (0.00 sec)

MySQL [(none)]> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
| threetierdb |
+-----+
5 rows in set (0.01 sec)

MySQL [(none)]>
```

- Create a **app.js** file and edit with below mentioned code;

```
const mysql = require('mysql2');

const connection = mysql.createConnection({
  host: 'your-db-instance.xxxxxx.us-east-1.rds.amazonaws.com',
  user: 'your-username',
  password: 'your-password',
  database: 'your-database',
});

connection.connect((err) => {
  if (err) {
    console.error('Error connecting to the database:', err.message);
    return;
  }
  console.log('Connected to the MySQL database!');
  connection.end(); // Close the connection after testing
});
```

Note : Replace corresponding

host : DB endpoint (got from terraform output)

username : DB username

password : DB password

database : Database name

- Run the app.js by
node app.js

```
[ec2-user@ip-10-0-3-94 ~]$ node app.js
Connected to the MySQL database!
[ec2-user@ip-10-0-3-94 ~]$
```

The connection was "Successful"...!!!!

Then the second phase of the project is there; transition to **Terraform Cloud CI/CD** by removing the S3 remote backend and DynamoDB state lock in Favor of Terraform Cloud's built-in capabilities.

Terraform Cloud CICD

Prepare Your Terraform Cloud Account

Organization:

Sign up/Login:

- Create or log into your Terraform Cloud account at Terraform Cloud.

Create an Organization:

- Navigate to the "Organizations" tab and create a new organization if you don't already have one.

Workspace:

Create a Workspace:

- Go to your organization and select "Workspaces."
- Click "**New Workspace**" and choose "**Version control workflow**".

Connect to Your GitHub Repository:

- Authenticate Terraform Cloud with GitHub if not already done.
- Select the repository containing your Terraform configuration files.

Specify Workspace Settings:

- Name your workspace (e.g., AWSTerraform_ThreeTierArch).
- Set the **Working Directory** to the folder containing your Terraform configuration files if it's not in the root of the repository.
- Enable **Automatic Runs** to automatically detect changes on git push.

Create a new Workspace

HCP Terraform organizes your infrastructure resources by workspaces. A workspace contains infrastructure resources, variables, state data, and run history. [Learn more](#) about workspaces in HCP Terraform.

Choose your workflow

Version Control Workflow

Trigger runs based on changes to configuration in repositories.



Best for those who need traceability and transparency

CLI-Driven Workflow

Trigger runs in a workspace using the Terraform CLI.



Best for those comfortable with Terraform CLI

API-Driven Workflow

Trigger runs using the HCP Terraform API.



Best for those with custom integrations and pipelines

Connect to Your GitHub Repository

HCP Terraform organizes your infrastructure resources by workspaces. A workspace contains infrastructure resources, variables, state data, and run history. [Learn more](#) about workspaces in HCP Terraform.



Connect to VCS



Choose a repository



Configure settings

Connect to a version control provider

Choose the version control provider that hosts the Terraform configuration for this workspace.

Project: Default Project

GitHub



GitHub App

[Connect to a different VCS](#)

Create a new Workspace

HCP Terraform organizes your infrastructure resources by workspaces. A workspace contains infrastructure resources, variables, state data, and run history. [Learn more](#) about workspaces in HCP Terraform.

Configure Settings

Workspace Name

AWSTerraform_ThreeTierArch

The name of your workspace is unique and used in tools, routing, and UI. Dashes, underscores, and alphanumeric characters are permitted. [Learn more about naming workspaces](#)

Description (Optional)

Workspace description

Advanced options

[< Previous](#)

Cancel

Create

First add your **AWS_ACCESS_KEY_ID** and **AWS_SECRET_ACCESS_KEY** variables for your IAM user with admin permissions. Ensure that you click Sensitive for these variables. This will prevent your variable value from being displayed.

You'll also need to create a variable for **AWS_DEFAULT_REGION** set to "us-east-1". Also create a **CONFIRM_DESTROY** variable set to **1**. This is needed to destroy our infrastructure later.

Configure Backend for Remote State

Update the terraform block in your configuration files to use Terraform Cloud's backend.

```
terraform {  
  cloud {  
    organization = "CloudAravind"  
  
    workspaces {  
      name = "three-tier-architecture"  
    }  
  }  
}
```

Save and push the changes to the GitHub repository.

Test and Initialize Terraform Cloud

Trigger a Run:


- Go to your Terraform Cloud workspace and verify if a run is triggered automatically on git push.
- If not, click "**Start New Plan**" to manually trigger it.


Review the Plan:

- Check the proposed changes and ensure they match expectations.

Apply the Plan:

- Approve and apply the plan directly from Terraform Cloud.

 **aravindur** triggered a **run** from UI a few seconds ago Run Details


 **Plan finished** a few seconds ago

Resources: **44** to add, **0** to change, **0** to destroy


Started a few seconds ago > Finished a few seconds ago

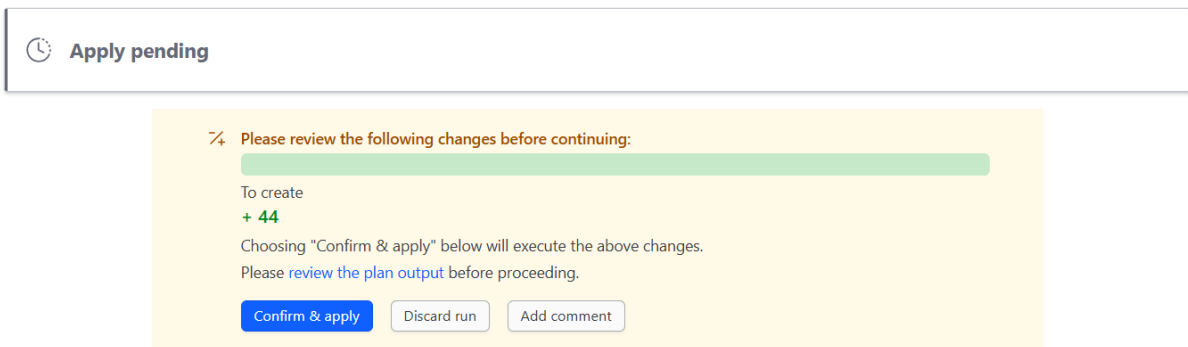
+ 44 to create

Filter resources by address...

 Filter by action

Terraform 1.10.3

 Download raw log



Approve and apply the plan directly from Terraform Cloud.

Verify CI/CD Workflow

Although not the main intention of the project, we have also set up our GitHub to kick off our pipeline in Terraform Cloud. To test this you can clone this repo, make a change, then push back to your GitHub. Then switch over to the Terraform Cloud Workspace and see that there is a current run in planning. You will still need to Confirm & Apply for changes to take effect.

Modify Configuration:

- Make a change in the configuration file (e.g., update an EC2 instance type) and commit it to the GitHub repository.

Observe Terraform Cloud:

- Ensure a new run is triggered automatically in the workspace upon git push.
- Review the proposed changes and apply them.

This setup ensures that your Terraform Cloud workspace detects configuration changes automatically, runs terraform plan, and allows you to apply changes to maintain your AWS three-tier architecture efficiently.

Clean Up

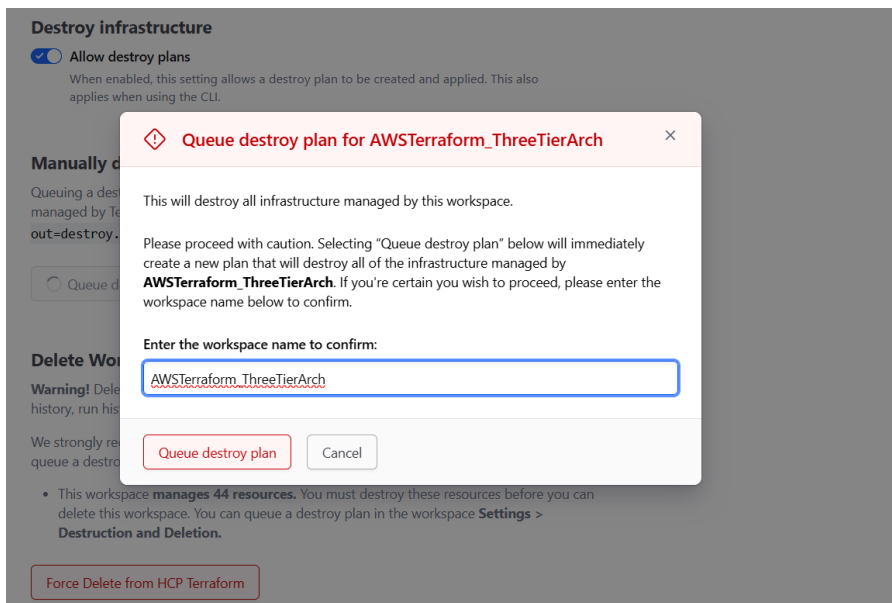
- Navigate to your Workspace and click **Settings > Destruction and Deletion**.
- Click **Queue destroy plan**.

Manually destroy

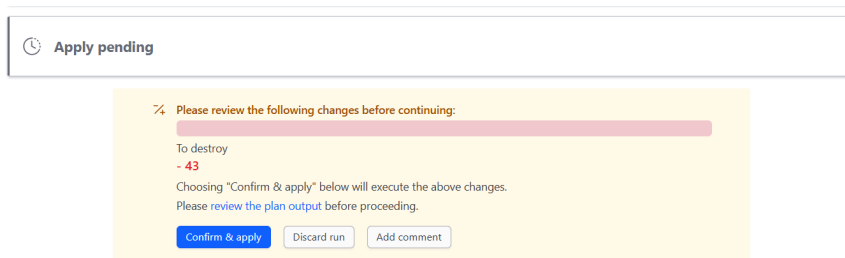
Queuing a destroy plan will redirect to a new plan that will destroy all of the infrastructure managed by Terraform. It is equivalent to running `terraform plan -destroy -out=destroy.tfplan` followed by `terraform apply destroy.tfplan` locally.

Queue destroy plan

- You will need to type the name of your Workspace then click **Queue destroy plan**.



- Once the Plan is finished click **Confirm & Apply**.



GitHub Repo is [here](#)

The project has been successfully completed!!!

