**Infrastructure Creation using Terraform Modules with OIDC Integration and Automation with GitHub Action**

**Project Summary:**

In this project, we're setting up a secure and automated infrastructure using Terraform modules, OIDC integration, and GitHub Actions. The goal is to create a modular and reusable codebase that simplifies the management of resources like VPCs, Security Groups, and EC2 instances. By integrating OIDC, we ensure secure authentication for our applications. GitHub Actions automates the deployment process, making it consistent and efficient. Overall, this project aims to provide a scalable and maintainable infrastructure setup that enhances security and automation.

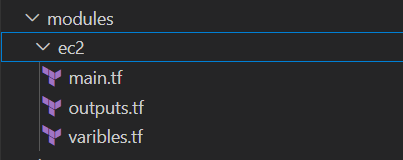
**why modules in Terraform:**

Modules in Terraform help you organize and group related resources together. This makes our infrastructure code easier to manage, reuse, and maintain. By using modules,

we can:

* Simplify Management: Keep your code organized and easy to understand.
* Reuse Code: Use the same module in different parts of your project without rewriting code.
* Maintain Consistency: Ensure that similar resources are set up in the same way every time.

**modules/ec2/main.tf**



**What:** This file defines the EC2 instances.

**Why:** It encapsulates the configuration for EC2 instances, making it reusable and modular.

**How:** By specifying the instance type, AMI, and other parameters, it creates EC2 instances as defined.

**modules/ec2/outputs.tf**

**What:** This file defines the outputs for the EC2 module.

**Why:** Outputs provide information about the resources created, which can be used by other modules or for reference.

**How:** It specifies the values to be output, such as instance IDs, making them accessible to other parts of the configuration.

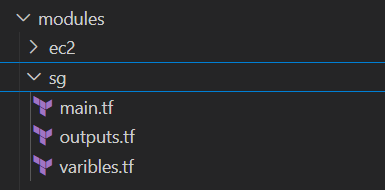
**modules/ec2/variables.tf**

**What:** This file defines the variables for the EC2 module.

**Why:** Variables allow for parameterization and flexibility in the module, enabling different configurations without changing the code.

**How:** By defining variables, it allows users to pass different values when calling the module.

**modules/sg/main.tf**



**What:** This file defines the security groups.

**Why:** It encapsulates the configuration for security groups, making it reusable and modular.

**How:** By specifying ingress and egress rules, it creates security groups with the defined rules.

**modules/sg/outputs.tf**

**What:** This file defines the outputs for the security group module.

**Why:** Outputs provide information about the resources created, which can be used by other modules or for reference.

**How:** It specifies the values to be output, such as security group IDs, making them accessible to other parts of the configuration.

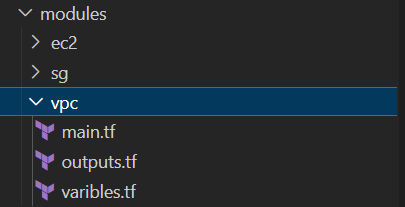
**modules/sg/variables.tf**

**What:** This file defines the variables for the security group module.

**Why:** Variables allow for parameterization and flexibility in the module, enabling different configurations without changing the code.

**How:** By defining variables, it allows users to pass different values when calling the module.

**modules/vpc/main.tf**



**What:** This file defines the VPC.

**Why:** It encapsulates the configuration for the VPC, making it reusable and modular.

**How:** By specifying the CIDR block and other parameters, it creates a VPC as defined.

**modules/vpc/outputs.tf**

**What:** This file defines the outputs for the VPC module.

**Why:** Outputs provide information about the resources created, which can be used by other modules or for reference.

**How:** It specifies the values to be output, such as VPC IDs, making them accessible to other parts of the configuration.

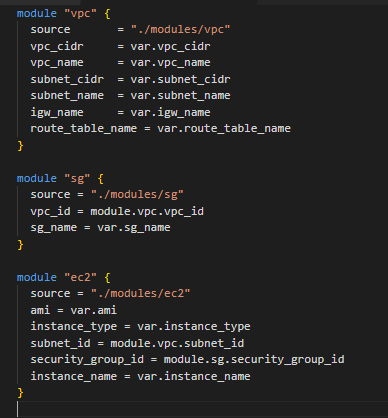
**modules/vpc/variables.tf**

**What:** This file defines the variables for the VPC module.

**Why:** Variables allow for parameterization and flexibility in the module, enabling different configurations without changing the code.

**How:** By defining variables, it allows users to pass different values when calling the module.

**Calling the modules using: main.tf**

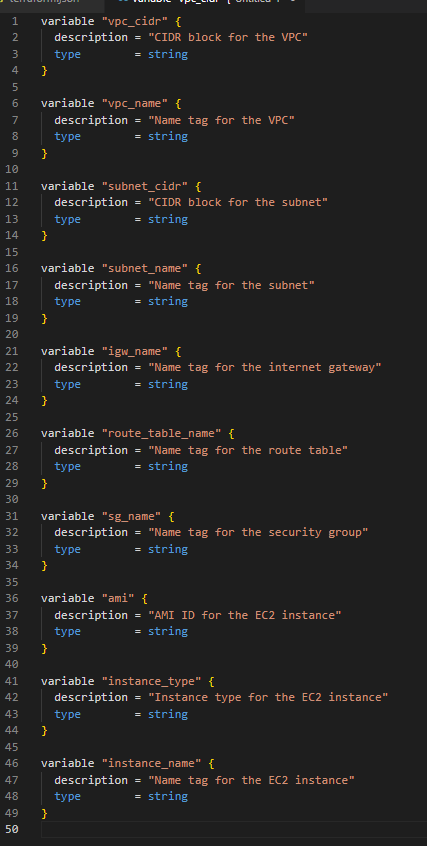


**What:** This is the main configuration file where you define your resources and call modules.

**Why:** It serves as the entry point for your Terraform configuration, organizing and managing the infrastructure resources.

**How:** By defining resources and modules, it orchestrates the creation and management of infrastructure components.

**variables.tf**

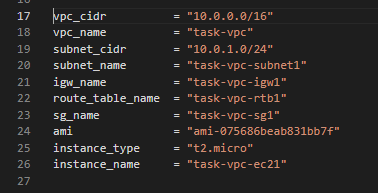


**What:** This file defines the variables for the main configuration.

**Why:** Variables allow for parameterization and flexibility in the configuration, enabling different configurations without changing the code.

**How:** By defining variables, it allows users to pass different values when running the Terraform configuration.

**terraform.tfvars**

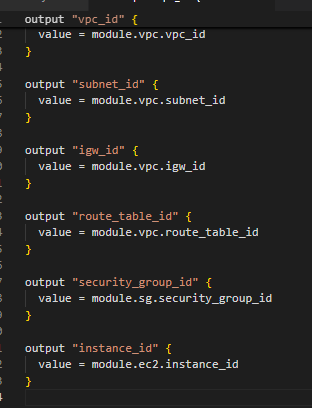


**What:** This file contains the values for the variables.

**Why:** It allows users to specify values for variables without changing the code, making the configuration flexible and reusable.

**How:** By defining variable values, it provides the necessary inputs for the Terraform configuration.

**outputs.tf**

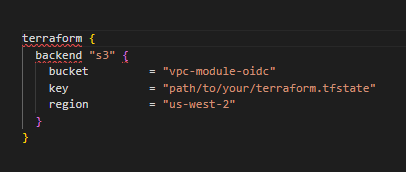


**What:** This file defines the outputs for the main configuration.

**Why:** Outputs provide information about the resources created, which can be used by other modules or for reference.

**How:** It specifies the values to be output, such as VPC IDs, instance IDs, and security group IDs, making them accessible to other parts of the configuration.

**Backend.tf**



**What:** This file configures the backend for Terraform, such as remote state storage.

**Why:** Using a remote backend allows multiple team members to work on the same Terraform configuration without conflicts, as the state is stored in a central location.

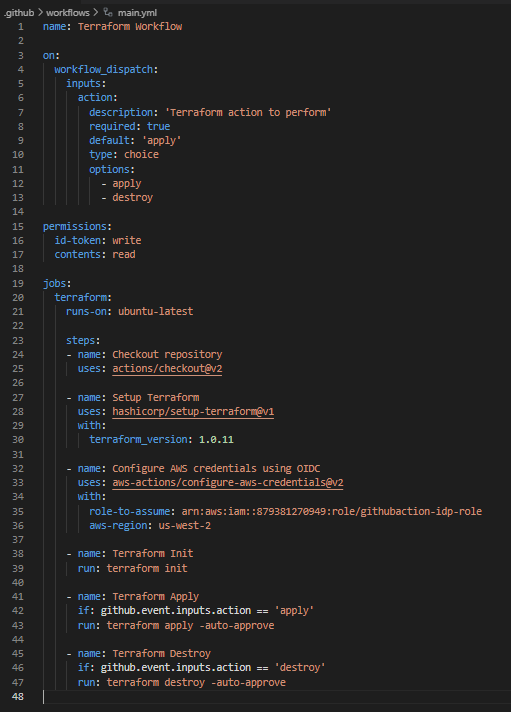
**How:** It specifies the backend type (e.g., S3) and its configuration details, ensuring that Terraform state is stored and managed remotely.

**README.md**

This file provides an overview of the project, including installation instructions, usage examples, and any other relevant information, it helps users understand the purpose of the project, how to set it up, and how to use it. By reading the README, users can quickly get up to speed with the project without needing to dive into the code.

Workflow Automation:

**Vpc-module-OIDC/.github/workflows/main.yml**



**---------------------------------------------------------------------**

Workflow Name

name: Terraform Workflow

**What:** This sets the name of the workflow.

**Why?** The name helps identify the workflow when you view it in the GitHub Actions interface. Naming workflows allows you to quickly distinguish between different workflows if you have multiple running for different purposes (e.g., build, deploy, test, etc.).

**How:**

The name field is a simple string that defines what you will see in the GitHub Actions interface. Here, we've set it to Terraform Workflow, but it can be anything that reflects the purpose of the workflow.

This name will show up on the GitHub Actions dashboard and also in the status checks on pull requests, helping everyone involved understand the workflow's purpose without opening it.

Trigger

on:

workflow\_dispatch:

inputs:

action:

description: 'Terraform action to perform'

required: true

default: 'apply'

type: choice

options:

- apply

**What:**

This part defines the event that triggers the workflow. In this case, the workflow is triggered manually via the workflow dispatch event, and users are asked to select an action (apply or destroy) when they trigger it.

**Why?**

workflow dispatch allows manual triggering of the workflow, which is useful in scenarios where you need control over when the workflow runs. You might want to trigger the Terraform workflow only after reviewing code or after a specific event (like a pull request merge). The options apply and destroy allow the user to decide whether they want to apply infrastructure changes or tear down existing infrastructure.

**How:**

on: defines the event that triggers the workflow. In this case, workflow\_dispatch is used, which means the workflow can be manually triggered from the GitHub Actions interface.

inputs: allows you to define custom inputs for the user to choose when triggering the workflow.

The action input is defined as a choice with two options: apply or destroy.

The required: true ensures that the user must select an option before triggering the workflow, preventing accidental runs.

The default: 'apply' ensures that, if the user doesn't choose anything, the default action will be applying. Permissions

**permissions:**

id-token: write

contents: read

**What:** This sets the permissions for the workflow.

**Why?**  The id-token: write permission is required for OpenID Connect (OIDC) authentication to AWS, allowing the workflow to securely authenticate without long-term AWS credentials. The contents: read permission grants the workflow read access to the repository’s contents, which is necessary to access the Terraform code for execution.

**How:**

permissions: defines the set of permissions granted to the workflow during execution.

id-token: write allows the workflow to use OIDC authentication. With this permission, GitHub Actions can generate a temporary token that the workflow can use to authenticate securely with AWS, avoiding the need for stored AWS credentials.

contents: read allows the workflow to access files within the repository. Since Terraform configurations (like .tf files) are stored in the repository, the workflow needs this permission to read and apply the configurations.

**Job Definition**

Job Definition

jobs:

terraform:

runs-on: ubuntu-latest

**What:** This defines a job named terraform that runs on the ubuntu-latest virtual environment.

**Why?** Jobs specify the set of steps that GitHub Actions will run. The runs-on key ensures that the job runs on a consistent environment (in this case, the latest version of Ubuntu) across all executions. This prevents issues with inconsistent environments that might arise if different machines or OS versions were used.

**How:**

jobs: defines the set of jobs to run within the workflow.

terraform: is the name of this particular job. You can define multiple jobs in a workflow, but in this case, we’re just defining one job focused on Terraform execution.

runs-on: Ubuntu-latest specifies the operating system for the job. Ubuntu-latest refers to the most current version of Ubuntu available in the GitHub Actions runners. Other choices could include macos-latest or windows-latest, but we’re using Ubuntu here because it’s common for Terraform workflows.

Steps

Checkout Repository

steps:

- name: Checkout repository

uses: actions/checkout@v2

**What:** This step checks out the code from the repository to the runner.

**Why:** For subsequent steps to access the repository's files, it’s crucial to first fetch the latest version of the code. Without this, the workflow wouldn't have access to any Terraform files (or any code) in the repository.

**How:**

steps: defines a sequence of operations that will be executed as part of the job.

- name: Checkout repository provides a human-readable name for this step, making the workflow easier to understand.

uses: actions/checkout@v2 uses a predefined GitHub Action (actions/checkout@v2) that automatically clones the repository's contents to the GitHub Actions runner's workspace, making all files available to the workflow.

Setup Terraform

- name: Setup Terraform

uses: hashicorp/setup-terraform@v1

with:

terraform\_version: 1.0.11

**What:** This step installs Terraform on the runner.

**Why:** To use Terraform within the workflow, it must be installed on the runner. Specifying the version ensures that the workflow uses a consistent version of Terraform (in this case, 1.0.11), preventing discrepancies between local and CI environments.

**How:**

- name: Setup Terraform provides a label for the step.

uses: hashicorp/setup-terraform@v1 specifies that the action setup-terraform from HashiCorp should be used. This action handles the installation of Terraform.

with: allows additional parameters for the action.

terraform\_version: 1.0.11 specifies the exact version of Terraform to install, ensuring that the workflow uses the specified version and not a newer (or incompatible) one.

Configure AWS Credentials using OIDC

- name: Configure AWS credentials using OIDC

uses: aws-actions/configure-aws-credentials@v2

with:

role-to-assume: arn:aws:iam::879381270949:role/githubaction-idp-role

aws-region: us-west-2

**What:**

This step configures the workflow to authenticate with AWS using OpenID Connect (OIDC) and assumes an IAM role to obtain the necessary AWS credentials for the workflow to interact with AWS services.

**Why?**

When interacting with AWS services in a CI/CD pipeline (like this GitHub Actions workflow), securely managing AWS credentials is crucial. Rather than storing long-lived AWS credentials (like access keys and secret keys) in the repository or environment variables, OIDC provides a way for the workflow to securely authenticate with AWS and assume a role, temporarily granting it the necessary permissions to perform actions. This approach reduces the risk of leaking sensitive credentials.

Using OIDC also avoids the manual management of AWS credentials and ensures that credentials are valid only for the duration of the workflow run, improving security.

**How:**

Defining the Action

uses: aws-actions/configure-aws-credentials@v2

This tells GitHub Actions to use the aws-actions/configure-aws-credentials action to set up AWS credentials. The version @v2 ensures that the workflow uses the second major version of the action, which includes fixes and improvements over the first version.

The aws-actions/configure-aws-credentials action is a GitHub-provided action specifically designed to securely configure AWS credentials using OIDC. This is preferred over using AWS access keys, as it leverages a more secure, temporary authentication mechanism.

Providing Configuration Parameters

with: section specifies the configuration parameters required by the action:

role-to-assume: arn:aws:iam::879381270949:role/githubaction-idp-role

This is the Amazon Resource Name (ARN) of the IAM role the GitHub Actions workflow needs to assume to gain the necessary AWS permissions.

The IAM role defined here (githubaction-idp-role) must be pre-configured with permissions that allow the workflow to perform actions like creating or destroying infrastructure (e.g., managing EC2 instances, VPCs, or S3 buckets).

This IAM role is set up with trust policies that allow GitHub Actions to authenticate via OpenID Connect and assume the role.

By using OIDC, you don’t need to store long-term AWS credentials (like an access key and secret key). Instead, GitHub Actions uses a temporary security token, generated based on the OIDC authentication, to assume the role and gain permissions for AWS actions.

The role you specify (arn:aws:iam::879381270949:role/githubaction-idp-role) must have the correct permissions to allow the workflow to carry out tasks, like applying or destroying Terraform-managed infrastructure.

aws-region: us-west-2

This specifies the AWS region in which the resources managed by the Terraform workflow will be provisioned or destroyed.

The region is necessary because AWS services are region-specific, and operations like creating EC2 instances or modifying VPCs need to be performed within a particular region.

In this case, the us-west-2 region (which corresponds to Oregon) is specified. You can change this to another region (e.g., us-east-1 or ap-south-1) based on your needs.

**How OIDC Works in This Context**

OIDC Authentication: OpenID Connect (OIDC) is a secure identity protocol that allows external systems (like GitHub Actions) to authenticate with AWS. Instead of using long-lived access keys, OIDC allows the GitHub Actions runner to authenticate directly with AWS through the GitHub-OIDC integration.

When the workflow runs, GitHub Actions sends an OIDC token to AWS. AWS verifies the token against a pre-configured trust policy in IAM and, if the authentication is valid, the workflow is allowed to assume the IAM role and gain temporary security credentials.

These temporary credentials allow the workflow to interact with AWS services without ever needing to store or manage long-term credentials.

Security Considerations

Temporary Credentials: The credentials obtained via OIDC are temporary and expire after a short duration (typically an hour). This reduces the risk of credential exposure or misuse.

No Secrets in the Codebase: The key benefit of using OIDC is that you avoid having sensitive AWS access keys stored in the repository, CI/CD environment, or elsewhere. This significantly improves the security posture of your workflow.

Trust Relationship in IAM: The IAM role githubaction-idp-role should have a trust relationship policy set up to allow GitHub to authenticate using OIDC. This ensures that only authenticated requests from GitHub Actions can assume the role.

-------------------------------------------------------------------------------------------------------------------------------

Terraform Init

- name: Terraform Init

run: terraform init

**What:** This step initializes the Terraform configuration.

**Why:** Initialization is required to set up the backend and download necessary provider plugins. How:

**- name:** Terraform Init gives a name to the step.

**run:** terraform init specifies the command to run. The terraform init command initializes the working directory containing the Terraform configuration files. This step sets up the backend configuration and downloads the necessary provider plugins, preparing the environment for Terraform operations4.

Terraform Apply

- name: Terraform Apply

if: github.event.inputs.action == 'apply'

run: terraform apply -auto-approve

**What:** This step applies the Terraform configuration if the selected action is apply.

**Why:** It creates or updates the infrastructure as defined in the Terraform configuration.

**How:**

**- name:** Terraform Apply gives a name to the step.

if: github.event.inputs.action == 'apply' specifies a condition. This step will only run if the action input is apply.

run: terraform apply -auto-approve specifies the command to run. The terraform apply -auto-approve command applies the Terraform configuration without requiring manual approval. This step executes the Terraform plan and applies the changes to create or update the infrastructure.

Terraform Destroy

- name: Terraform Destroy

if: github.event.inputs.action == 'destroy'

run: terraform destroy -auto-approve

**What:** This step destroys the Terraform-managed infrastructure if the selected action is destroy. **Why:** It removes the infrastructure, useful for cleanup or decommissioning resources.

**How:**

**- name:** Terraform Destroy gives a name to the step.

if: github.event.inputs.action == 'destroy' specifies a condition. This step will only run if the action input is destroy.

run: terraform destroy -auto-approve specifies the command to run. The terraform destroy -auto-approve command destroys the Terraform-managed infrastructure without requiring manual approval. This step removes the resources defined in the Terraform configuration.