Day 4: Terraform Modules and Reusability

Introduction to Modules:

- Terraform modules are a fundamental concept for creating reusable and maintainable infrastructure as code.
- Modules allow you to encapsulate a set of resources and their configurations into a reusable package.
- This modular approach promotes code organization, reduces duplication, and enhances collaboration among team members.

Key concepts of Terraform modules include:

1. Root module: The main configuration files in your working directory.

2. Child modules: Separate configurations that can be called by the root module or other modules.

3. Module sources: Where modules are stored (local paths, Git repositories, Terraform Registry, etc.).

4. Input variables: Parameters that allow customization of module behavior.

5. Output values: Data that a module can expose for use by the calling module.

Creating reusable modules for common infrastructure patterns:

When creating modules, focus on encapsulating logical groups of resources that are commonly used together. For example, you might create modules for:

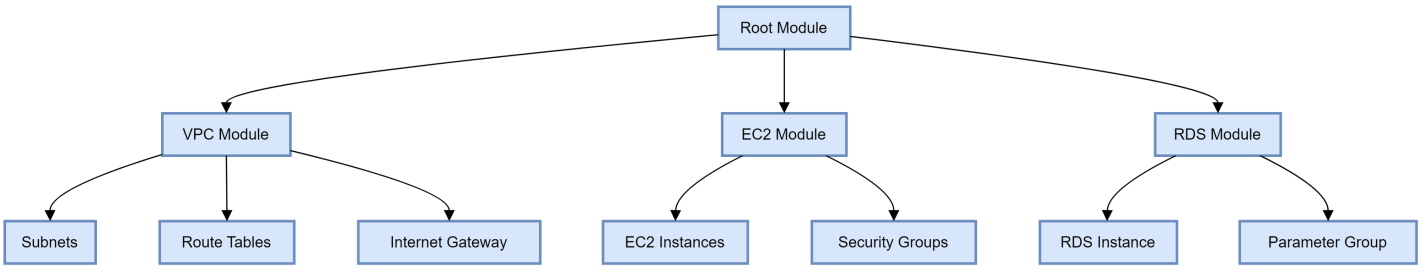
1. Networking (VPC, subnets, route tables)

2. Security groups

3. Application load balancers

4. Database clusters

Example of a simple module structure for a VPC:



This diagram illustrates a typical module structure with a root module calling child modules for VPC, EC2, and RDS resources.

Working with module inputs, outputs, and versions:

1. Input variables:

- Defined in `variables.tf`

- Allow customization of module behavior

- Example:

```
variable "vpc_cidr" {
 description = "CIDR block for the VPC"
 type = string
 default = "10.0.0.0/16"
}
```

2. Output values:

- Defined in `outputs.tf`

- Expose data for use by the calling module

- Example:

```
output "vpc_id" {
 description = "ID of the created VPC"
 value = aws_vpc.main.id
}
```

3. Module versions:

- Use semantic versioning (e.g., v1.2.3)

- Specify versions in module source to ensure consistency

- Example:

```
module "vpc" {
 source = "terraform-aws-modules/vpc/aws"
 version = "3.14.0"
 # Module inputs
 name = "my-vpc"
 cidr = var.vpc_cidr
}
```

Industry Insight:

How companies use modules for standardization and consistency across teams:

1. **Shared module repositories**: Many organizations maintain internal repositories of approved, tested modules that teams can use across projects. This ensures consistency and reduces duplication of effort.

2. **Module governance:** Companies often implement processes to review and approve modules before they're added to the shared repository. This helps maintain quality and security standards.

3. **Infrastructure templates**: Some organizations create complete infrastructure templates using a combination of modules. These templates can be used as starting points for new projects, ensuring best practices are followed.

4. **Versioning strategies**: Companies typically adopt strict versioning policies for their modules to manage changes and ensure backwards compatibility.

5. Documentation: Well-documented modules are crucial for adoption across teams. Many companies use tools like Terraform-docs to generate consistent documentation for their modules.

Real-world scenario:

A large e-commerce company uses a modular approach to manage its cloud infrastructure across multiple regions and environments (dev, staging, production). They have a set of core modules for:

- Networking (VPC, subnets, security groups)

- Compute (EC2 instances, auto-scaling groups)

- Database (RDS instances, read replicas)

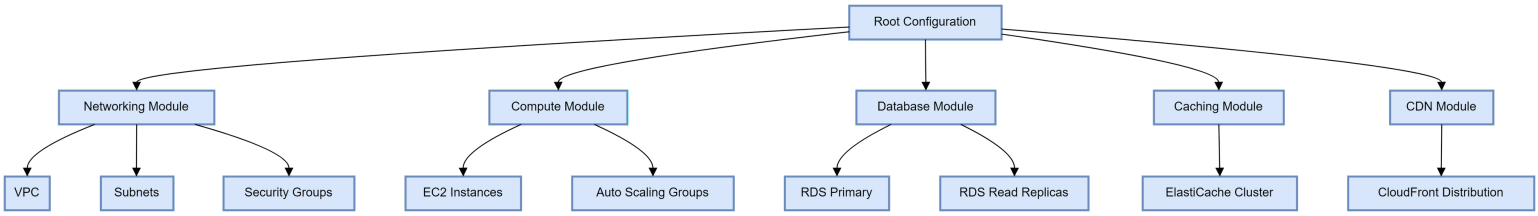
- Caching (ElastiCache clusters)

- Content Delivery (CloudFront distributions)

These modules are versioned and stored in a private Git repository.

When a new project or environment needs to be set up, the team uses a combination of these modules to quickly deploy a standardized infrastructure.

This approach has significantly reduced the time to set up new environments and has improved consistency across the organization's infrastructure.



This diagram shows how different modules can be combined in a root configuration to create a complete e-commerce infrastructure.

Exploring Terraform Registry and internal module registries:

1. **Terraform Registry (registry.terraform.io):**

- Public repository of Terraform modules

- Provides versioned, documented modules for various providers

- Easy to use with the `source` attribute in module blocks

2. **Internal module registries:**

- Private repositories hosted within an organization

- Can be set up using tools like Terraform Enterprise or open-source alternatives

- Allows for better control and governance of modules used within the company

Hands-On Lab:

Building a custom module and using it in a project

we'll create a simple AWS VPC module and use it in a root configuration.

File: modules/vpc/main.tf

```
resource "aws_vpc" "main" {
 cidr_block = var.vpc_cidr
 enable_dns_hostnames = true
 enable_dns_support = true
 tags = {
 Name = var.vpc_name
 }
}
resource "aws_internet_gateway" "main" {
 vpc_id = aws_vpc.main.id
  tags = {
 Name = "${var.vpc_name}-igw"
 }
}
 resource "aws_subnet" "public" {
 count = length(var.public_subnet_cidrs)
 vpc_id = aws_vpc.main.id
 cidr_block = var.public_subnet_cidrs[count.index]
 availability_zone = var.availability_zones[count.index]
  tags = {
 Name = "${var.vpc_name}-public-subnet-${count.index + 1}"
 }
}
 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
 }
 tags = {
 Name = "${var.vpc_name}-public-rt"
 }
}
 resource "aws_route_table_association" "public" {
 count = length(aws_subnet.public)
 subnet_id = aws_subnet.public[count.index].id
 route_table_id = aws_route_table.public.id
}
# File: modules/vpc/variables.tf
variable "vpc_cidr" {
 description = "CIDR block for the VPC"
 type = string
}
variable "vpc_name" {
 description = "Name of the VPC"
 type = string
}
variable "public_subnet_cidrs" {
 description = "List of public subnet CIDR blocks"
 type = list(string)
```

```
}
variable "availability_zones" {
 description = "List of availability zones"
 type = list(string)
}
# File: modules/vpc/outputs.tf
output "vpc_id" {
 description = "ID of the created VPC"
 value = aws_vpc.main.id
}
 output "public_subnet_ids" {
 description = "List of public subnet IDs"
 value = aws_subnet.public[*].id
}
 # File: main.tf (root configuration)
provider "aws" {
 region = "us-west-2"
}
module "vpc" {
 source = "./modules/vpc"
 vpc_cidr = "10.0.0.0/16"
 vpc_name = "my-custom-vpc"
 public_subnet_cidrs = ["10.0.1.0/24", "10.0.2.0/24"]
 availability_zones = ["us-west-2a", "us-west-2b"]
}
```

```
output "vpc_id" {
 value = module.vpc.vpc_id
}
output "public_subnet_ids" {
 value = module.vpc.public_subnet_ids
}
```

To use this module:

1. Create a directory structure:

```
your_project/
 ├── modules/
 │   └── vpc/
 │   ├── main.tf
 │   ├── variables.tf
 │   └── outputs.tf
 └── main.tf
```

2. Copy the code from the artifact into the respective files.

3. Initialize Terraform:

- terraform init

4. Review the plan:

- terraform plan

5. Apply the configuration:

- **terraform apply**

This will create a VPC with two public subnets and an Internet Gateway using your custom module.

1. What is the primary purpose of using Terraform modules?

a) To create reusable and shareable components of infrastructure

b) To increase the complexity of Terraform configurations

c) To reduce the number of resources that can be created

d) To eliminate the need for variables in Terraform

2. How can you specify a specific version of a module from the Terraform Registry?

a) By using the `version` argument in the module block

b) By including the version number in the `source` URL

c) By setting a `module_version` variable

d) Modules from the Terraform Registry cannot be version-controlled

3. What file is commonly used to define input variables for a module?

a) inputs.tf

b) variables.tf

c) params.tf

d) config.tf

4. How can you reference an output value from a module in the root configuration?

a) ${module.module_name.output_name}

b) module.module_name.output_name

c) $(module.module_name.output_name)

d) output.module_name.output_name

5. Which of the following is NOT a best practice for creating Terraform modules?

a) Using input variables to make modules flexible

b) Providing clear documentation for module usage

c) Hardcoding credentials within the module

d) Using consistent naming conventions for resources

Answers:

1. a

2. a

3. b

4. b

5. c

This concludes the content for Day 4 on Terraform Modules and Reusability.

The structure follows the provided reference, including sections on industry insights, real-world scenarios, a hands-on lab, and exam practice questions.

The content is enhanced with Diagrams as Code to visualize key concepts.