Terraform Cheatsheet

A quick reference guide for Terraform, covering basic commands, resource definitions, modules, and best practices for infrastructure as code.

Terraform Basics

Core Concepts

Infrastructure as Code (IaC): Managing and provisioning infrastructure through code rather than manual processes.

Declarative Configuration: Defining the desired state of the infrastructure, and Terraform figures out how to achieve it.

State Management: Terraform tracks the state of your infrastructure to understand what resources it manages and how they relate to each other.

Providers: Plugins that allow Terraform to interact with various cloud providers (AWS, Azure, GCP) and other services.

Essential Commands

`terraform init`	Initializes a Terraform working directory. Downloads providers and modules.
`terraform plan`	Creates an execution plan, showing the changes Terraform will make.
`terraform apply`	Applies the changes required to reach the desired state of the configuration.
`terraform destroy`	Destroys all resources managed by the Terraform configuration.
`terraform show`	Inspect the current state.
`terraform output`	Show output values from the state.

Configuration Files

Terraform configuration files are written in HashiCorp Configuration Language (HCL) or JSON.

Files typically have a `.tf` extension.

A basic configuration includes `terraform`, `provider` and `resource` blocks.

Resources and Providers

Resource Definition

A `resource` block declares a resource of a given type (e.g., `aws_instance`) and a local name.

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b3c825232a0d4"
  instance_type = "t2.micro"
}
```

Attributes within the resource block configure the resource (e.g., `ami`, `instance_type`).

Provider Configuration

The `provider` block configures a specific provider, such as AWS, Azure, or GCP.

```
provider "aws" {
  region = "us-west-2"
}
```

Credentials for the provider can be configured through environment variables, or through the `profile` argument.

Data Sources

Data sources allow Terraform to fetch information about existing resources.

```
data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server/*"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }

  owners = ["099720109477"] # Canonical
}
```

Use data sources to dynamically retrieve values needed for resource configuration.

Modules and Variables

Module Definition

Modules are reusable Terraform configurations that encapsulate a set of resources.

Modules improve code organization and reusability.

```
module "ec2_instance" {
  source = "./modules/ec2"
  instance_type = "t2.micro"
  ami = "ami-0c55b3c825232a0d4"
}
```

Input Variables

Variables allow you to parameterize your Terraform configurations.

```
variable "instance_type" {
  type = string
  description = "EC2 instance type"
  default = "t2.micro"
}
```

Variables can be defined in `variables.tf` or passed via command-line arguments or environment variables.

Output Values

Outputs expose values from your Terraform configuration, making them accessible to other configurations or users.

```
output "instance_public_ip" {
  value = aws_instance.example.public_ip
  description = "The public IP of the EC2 instance."
}
```

Outputs are displayed after a successful `terraform apply`.

State Management and Best Practices

State Storage

Terraform state should be stored remotely for collaboration and consistency.

Common remote state backends include AWS S3, Azure Storage Account, and HashiCorp Terraform Cloud.

```
terraform {
  backend "s3" {
    bucket = "my-terraform-state-bucket"
    key    = "terraform.tfstate"
    region = "us-west-2"
  }
}
```

Terraform Cloud

HashiCorp Terraform Cloud provides collaboration, version control, and remote state management.

It allows teams to manage infrastructure changes in a controlled and auditable manner.

Consider using Terraform Cloud for team-based infrastructure management.

Best Practices

Version Control: Store your Terraform configurations in a version control system like Git.

Code Reviews: Use code reviews to ensure the quality and correctness of your Terraform configurations.

Testing: Implement automated testing to validate your infrastructure changes.

Idempotency: Ensure that running `terraform apply` multiple times produces the same result.

Locking: Remote state backends support locking, which prevents concurrent modifications to the state file.