

Terraform for Beginners

Portable IaC in the era of multi/hybrid cloud

Agenda

- Terraform for Beginners
- Setup
- Tutorials

Terraform for Beginners

- Why?
 - focus on cloud portability, although other providers (e.g. OC, K8s) exists
 - declarative language
 - conf diffs detected wrt saved conf state (of live infra)
 - no control flow constructs such as for-loop
 - additional providers can be [written](#) in Go as simple resource handlers
- GitOps
 - push-based deployment approach (see [Gitlab article](#))

Resources

- basic objects/entities to be managed;
- meta-arguments can be defined
 - `depends-on`,
 - `count` to create multiple instances of the same resource type,
 - `lifecycle` to define Terraform-related behavior such as upon update or deletion;

Modules

- grouping a set of resources into a reusable named component
- published and maintained as a whole;
- code reuse & more maintainable architecture;
- design pattern: separate code repo (modules) from live infrastructure;

Providers

- managers of specific resource types;
- providers are indexed on the [Terraform Registry](#)
- and can come from either Hashicorp, verified organizations or community members;
- No longer maintained ones are listed as "Archived".
- The [AWS Provider](#) is maintained directly by Hashicorp. The documentation is available [here](#) and the Github repo [here](#).

Miscellaneous

- **Input Variables** - used to abstract and parametrize providers;
- **Outputs** - specifying values to export from a module;
 - print to stdout when applying the configuration;
 - can be retrieved using the `terraform output <name>` command (e.g. `terraform output region`);
- **Data Sources** - defining a reference to information defined outside of Terraform;
- control flow: only if-else construct, to define multiple variants of the modeled infrastructure, by deploying either these or those resources based on data or variable values.

State management

- State is a persistent representation of the infrastructure
- Default is a local file (see example 1)
- **Problem:** Even if committed conflicts may arise if multiple Terraform runs are performed in parallel
- **Solution:** Use a remote state backend
- Multiple backends supported (GCS, S3, Azure Storage, Terraform Cloud, etc.)
- Using S3 backend in example 2
 - S3 is 99.99% available
 - supports server-side encryption using AES-256 and SSL-based communication
 - supports versioning so rolling back is possible
 - supports locking via DynamoDB

Multi-env management

Problem: variables not allowed in the backend block

- **Solution 1:** use partial configuration, i.e. move parameters to an env specific file recalled with `-backend-config <conf.hcl>`
- **Solution 2:** use workspaces (conceptually similar to git branching), each environment has a different managed state ending up in a different subfolder;

In practice

- Terraform often used through [Terragrunt](#) (thin wrapper around Terraform)
- following DRY principles
 - [Multi-env configuration without replicating code](#)
 - using separation in modules and variables `terragrunt.hcl` files
 - [Remote state \(backend\) configuration](#)
 - using `terragrunt.hcl` files to define state configuration in a remote state block
 - **automagically** creating the state store (S3, GCP) and the lock store (DynamoDB)
 - [run all modules](#) in subfolders in parallel
 - working with [multiple AWS accounts](#) by specifying IAM roles

Setup

Prerequisites

1. Install Terraform

- i. using a package manager
- ii. by downloading the binary from [here](#) or [here](#)

2. Decide where to deploy

i. AWS

- a. Sign up for AWS account: create non-root user and assign some policies
- b. Create a `~/.aws/credentials` file with a new profile for the account created at 1 or export `AWS_SECRET_ACCESS_KEY` and `AWS_ACCESS_KEY_ID`

ii. [Localstack](#)

Terraform project lifecycle

1. `terraform init` to initialize the Terraform project
2. `terraform plan` to see changes to the infrastructure wrt the applied tf file
3. `terraform apply` to apply the changes to the infrastructure (or `terraform apply -auto-approve` to skip confirmation)
4. Once done `terraform destroy` to terminate all resources managed by the current configuration;

Tutorials

Tutorials

1. Warm-up (no new resources added), local state file
2. Shared-state on S3
3. S3 bucket and Athena
4. Kinesis stream to S3 bucket
5. Python lambda function
6. Python lambda function on localstack