Infrastructure as Code (IaC) in the Cloud Age

Marek Wiewiórka, Tomasz Gambin

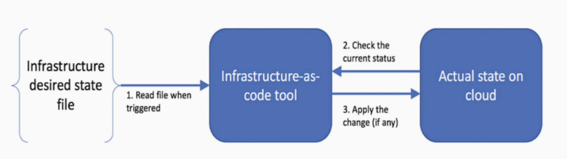
October 2024

Agenda

1. Intro
2. Infrastructure as Code principles and core concepts
3. Not too short introduction to Terraform

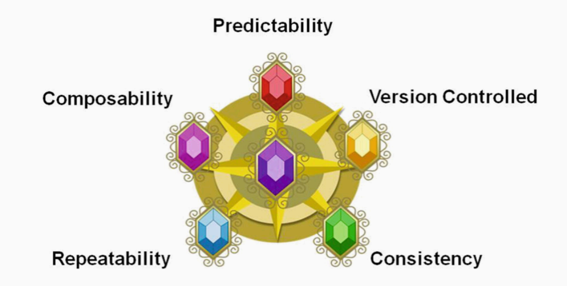
Infrastructure as Code (IaC) explained

- allows to write and execute code to define, deploy, update, and destroy your infrastructure
- gives rise to *mutable* infrastructure as the lifecycle of every infra resource,component is treated via code
- encourages declarative style of code wherein the desired end state and the configuration are present before final state is provisioned
- initially focusing on software, now also on virtualized hardware



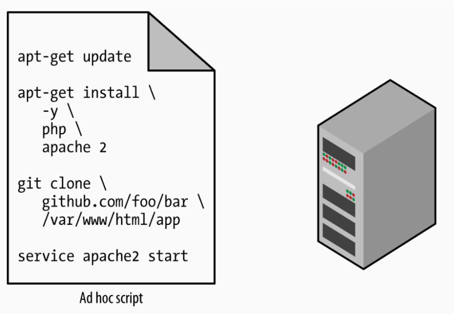
IaC principles

- **Version control** provides traceability of changes
- **Predictability** capability to always provide the same environment
- **Consistency** multiple instances of the same baseline code provide a similar environment
- **Composability** managed in a modular and abstracted format – *reusability*, speed and *safety* and automatic documentation

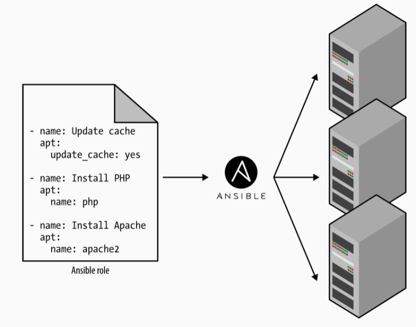


Categories of IaC tools 1/3

- Ad hoc scripts

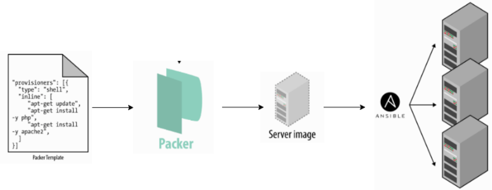


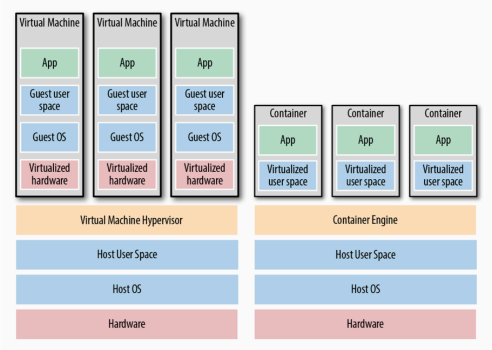
- Configuration management tools



Categories of IaC tools 2/3

- Server Templating Tools (e.g. Docker, Packer)





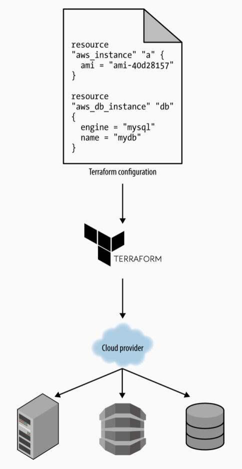
Categories of IaC tools 3/3

- Provisioning Tools (e.g. Terraform, Pulumi)

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}

resource "google_dns_record_set" "a" {
  name         = "demo.google-example.com"
  managed_zone = "example-zone"
  type         = "A"
  ttl          = 300
  rrdatas      = [aws_instance.example.public_ip]
}
```

- not only VM instances, also VPC (Networking), Managed Services (e.g. Dataproc), etc.



IaC differences - „*how*” vs „*what*”

- Configuration management versus provisioning
- Mutable infrastructure versus immutable infrastructure – *configuration drift* problem – mostly software layer – deployment in a form of an immutable *template* - e.g. Docker image, hard drive image
- Procedural language versus declarative language

- Ansible - imperative („*how*”)

```yaml
- ec2:
    count: 10
    image: ami-0c55b159cbfafe1f0
    instance_type: t2.micro
```

- Terraform - declarative („*what*”)

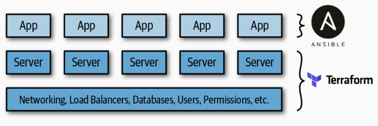
```hcl
resource "aws_instance" "example" {
  count         = 10
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Kubernetes ecosystem example - from Pod to SparkApplication

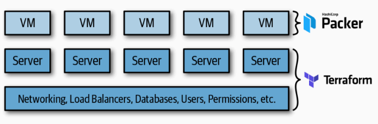
1. resource by resource with a `kubectl` command (e.g. create, run, scale) – imperative/low-level
2. Kubernetes Manifest file and `kubectl apply -f` – declarative (configuration builtin) but still low-level
3. Helm chart - versioning, templating (separation of configuration), reusability – declarative/ higher-level
4. Custom Resource Definition (CRD) and Kubernetes Operator (e.g. SparkOperator) – declarative/highest-level

Using Multiple Tools Together

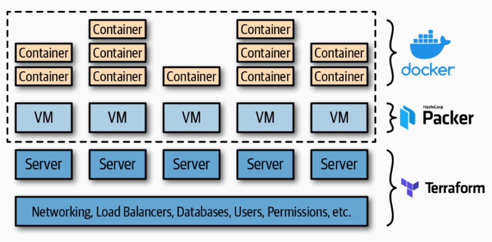
- Provisioning plus configuration management



- Provisioning plus server templating

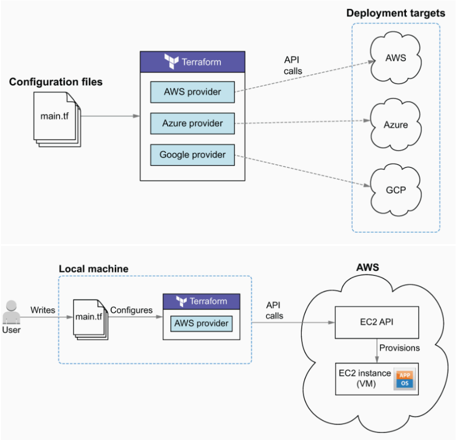


- Provisioning plus server templating plus orchestration (e.g. Google Kubernetes Engine)



Terraform - a provisioning tool

- cloud-agnostic
- open-source written in Golang
- cloud/services providers registry
- declarative programming HashiCorp Configuration Language (HCL)
- OpenTofu alternative



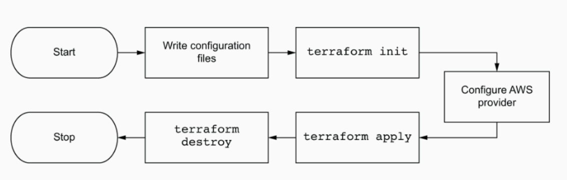
Terraform - a quick start

- macOS

```
brew tap hashicorp/tap
brew install hashicorp/tap/terraform
```

- Linux

```
curl -fsSL
↪ https://apt.releases.hashicorp.com/gpg
↪ | sudo apt-key add -
sudo apt-add-repository "deb
↪ [arch=amd64]
↪ https://apt.releases.hashicorp.com
↪ $(lsb_release -cs) main"
sudo apt-get update && sudo apt-get
↪ install terraform
```



Providers

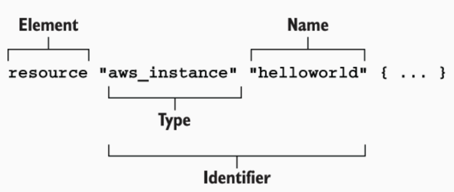
- provider - a plugin with a set of *resource* and *data* types that defines how changes to resources of that type are applied to remote APIs
- local utilities for tasks, like generating random numbers for unique resource names
- version constraints and semanting versioning

```
provider "google" {
  project = var.project_name
  region  = var.region
}
terraform {
  required_providers {
    google = {
      version = "~> 4.8.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.1.2"
    }
    kubectl = {
      source  = "gavinbunney/kubectl"
      version = "1.14.0"
    }
  }
}
```

Resources

- Each *resource* has inputs and outputs. Inputs are called *arguments*, and outputs are called *attributes*.
- attributes of resources can be referenced in other resources
- there are computed attributes that are only available after the resource has been created (e.g. cloud resource URLs or IDs)

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  [CONFIG ...]
}
```



Data sources

- represent a piece of read-only information that is fetched from the provider
- a way to query the provider's APIs for data and to make that data available to the rest of Terraform code.
- example use case - referencing Ubuntu image 22.04 (with updates)

Definition:

```
data "<PROVIDER>_<TYPE>" "<NAME>" {
  [CONFIG ...]
}
```

Referencing an attribute:

```
data.<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>
```

Variables 1/2

- input

```
variable "environment" {
  type        = string
  description = "Development or
  ↪ production environment"
  default = "dev"
  validation {
    condition     = contains(["dev",
    ↪ "prod"], var.environment)
    error_message = "Valid values
    ↪ for var: test_variable are
    ↪ (dev, prod)."
  }
}
```

- output

```
output "data_generator_lines_num" {
  value       =
  ↪ module.data-generator.lines_number
  description = "Number of lines in
  ↪ a generated file"
}
```

Variables 2/2

Passing input variables to a module:

- environment variables
  `TF_VAR_name`
- "*.tfvars"
- `terraform apply -var-file env/dev/project.tfvars`
- from command prompt
- default values

- local variables for modules

```
locals {
  service_name = "forum"
  owner        = "Community Team"
}
```

- help avoiding repeating the same values or expressions multiple times in a configuration

Implicit and explicit dependencies

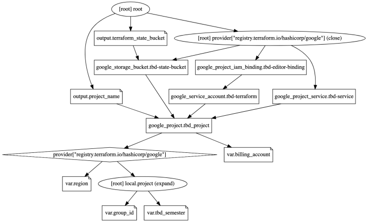
- implicit

```
resource "google_service_account" "tbd-sa"
↪ {
  account_id = "${var.project_name}-sa"
}

resource "google_project_iam_member"
↪  "tbd-sa-role-bindings" {
  project   = var.project_name
  role      = "roles/storage.admin"
  member    =
  ↪  "serviceAccount:${google_service_account.tbd-s
}
```

- explicit

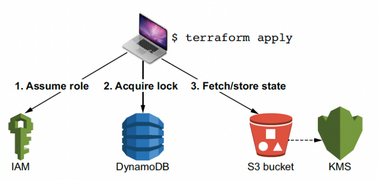
```
module "k8s-spark-operator"
↪  {
  depends_on = [module.gke]
  source     =
  ↪  "./modules/spark-on-k8s-op
}
```



Managing state

Locally(default):

- `terraform.tfstate` file in a JSON format
- error-prone
- not-secure



Shared storage

- requires defining a remote *backend* like S3, GCS
- encryption at rest and in transit(storing secrets)
- versioning
- isolation of environments using a bucket and/or prefix
- team collaboration

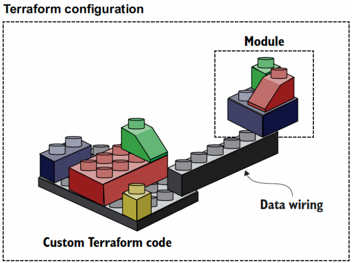
```
terraform init
↪ -backend-config=env/dev/backend.tfvar
↪ -reconfigure
```

Modules

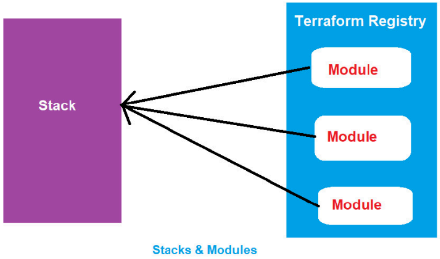
- any set of Terraform configuration files in a folder is a module
- there is always at least a *root* module
- code reusability
- can be stored locally or in a git repo
- versioning
- small, composable and testable

```
module "<NAME>" {
  source = "<SOURCE>"

  [CONFIG ...]
}
```



Modules vs. stacks

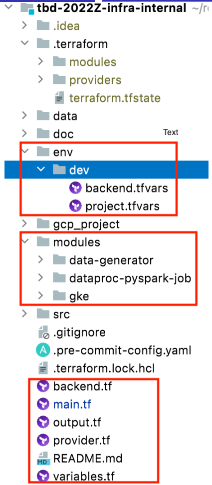


- *stacks* are collections of *modules* that are *logically* connected and defined in a single .tf file or multiple .tf files inside the same directory
- they represent a *single* deployment unit of an infrastructure, e.g. environment or a larger part of it, such as storage system

Benefits of using stacks

- limit *radius blast* of resource changes (separation of state files), i.e. human error boundaries
- speed – managing all resources with a single state file is slow
- different resource lifecycles – e.g. storage vs. compute layer
- separate management responsibilities across team boundaries

Project layout - a simple case



- `.terraform` scratch dir
- `env\/dev` with environment-specific variables
- `modules` local shared modules
- external git-hosted modules
- root module (stack) with `main.tf`
- state isolation per environment (limit radius blast and performance)

Loops

count

```
variable "subnet_ids" {
  type = list(string)
}
resource "aws_instance" "server" {
  # Create one instance for each subnet
  count = length(var.subnet_ids)

  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
  subnet_id     =
   ↪  var.subnet_ids[count.index]

  tags = {
    Name = "Server ${count.index}"
  }
}
```

- a change in the middle of the list ?

for_each

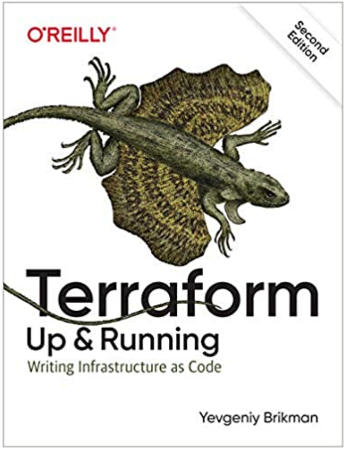
```
resource "google_project_iam_member"
 ↪  "tbd-editor-supervisors" {
  for_each = toset([
    "user:marek.wiewiorka@gmail.com",
    "user:tgambin@gmail.com"
  ])
  project =
   ↪  google_project.tbd_project.project_id
  role    = "roles/editor"
  member  = each.value
}
```

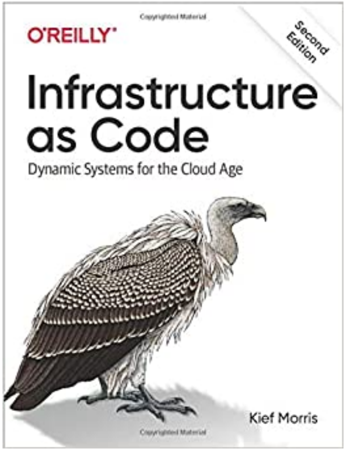
- set vs list updates

Tricks with expressions

- functions
- templating
- conditional expressions with ternary syntax (can be combined with `count` for optional modules)
- types and values
- list comprehensions with `for`
- *dynamic* blocks (within a resource or data type, e.g. configuration key-values)

Books


O'REILLY
Second Edition
Terraform
Up & Running
Writing Infrastructure as Code
Yevgeniy Brikman


O'REILLY
Second Edition
Infrastructure as Code
Dynamic Systems for the Cloud Age
Kief Morris