Terramate

Notes By Rohit Prakash
https://www.linkedin.com/in/rohit-prakash-204391252/

ABOUT TERRAMATE

Terramate is **a code generator and orchestrator for Terraform** that adds powerful capabilities such as code generation, stacks, orchestration, change detection, globals and more to Terraform

Infrastructure as Code is hard and complex - we get it!

Terramate helps you to overcome this by providing a ***next-generation*** Infrastructure as Code Management Platform that runs natively in your existing CI/CD system.

Using Terramate makes it possible to:

1. **Streamline configuration management and environments**: Terramate enables you to manage environments and services with stacks and adds code generation to keep configuration DRY, which means you can easily provision multiple services and environments and avoid manual copy & paste of duplicate configuration files.
2. **Automate and orchestrate Terraform, OpenTofu and Terragrunt in any CI/CD**: Terramate enables you to turn any CI/CD into a powerful IaC vending machine which means no more wasted time and money maintaining DIY solutions or buying expensive purpose-built CI/CD systems.
3. **Enable better collaboration, governance, observability and drift control**: Terramate adds a powerful suite of features to improve the developer experience when working with infrastructure as code such as plan previews, policies, observability, notifications, SlackOps, automated drift detection and reconciliation, which means better productivity, fewer failures and more stable infrastructure.

Terramate is designed and implemented by long-time Platform, Cloud and DevOps Engineering practitioners based on previous experience building and maintaining cloud infrastructure platforms for some of the most innovative companies in the world.

It can be integrated into any existing architecture and with every third-party tooling in less than 5 minutes, with no prerequisites, no lock-in and without modifying any of your existing configurations.

At the same time, Terramate integrates seamlessly and in a non-intrusive way with all your existing tooling, such as GitHub or Slack.

Terramate Features for Terraform Repositories

All Terramate features are now available to your team.

The following set of features highlights some special benefits:

- Use Terramate Change Detection to orchestrate Terraform in an efficient way
- Execute **any** command within stacks imported from terraform configuration.
- Run Terraform in any CI/CD following the Terramate Automation Blueprints and examples.
- Make use of Terramates advanced Code Generation and Globals to share data more easily.
- Synchronize deployments, drift runs, and previews to **Terramate Cloud** and get
  - Visibility of the Health of all Terraform Configurations over multiple repositories
  - Drift Detection in all Stacks
  - Pull Request Previews for actual changes
  - Notifications on deployment failures or newly detected drifts
  - Advanced collaboration and alert routing

Import Existing Terraform Stacks

To onboard Terramate to an existing Terraform project, you first need to import your Terraform root modules into Terramate.

```
terramate create --all-terraform
```

This command will detect existing Terraform root modules and create a stack configuration in them, which Terramate requires to identify a Terraform root module as a stack.

Run Terraform Commands

List all Stacks

```
Any Terramate CLI Feature is now available in your Stacks.
```

```
terramate list
```

Init Terraform

```
terramate run -- terraform init
```

Create a Terraform Plan in parallel

```
terramate run --parallel 5 -- terraform plan -out plan.tfplan
```

Apply a Terraform Plan in Changed Stacks

```
terramate run --changed -- terraform apply -auto-approve plan.tfplan
```

HOW IT IS USED IN PROJECT

Introduction

For a quick example of how Terramate works, this guide takes you through the following steps:

1. Creating a new Terramate project.
2. Adding stacks using Terramate CLI.
3. Generating a Terraform local backend configuration in all stacks using code generation.
4. Orchestrating commands such as terraform plan and terraform apply using orchestration and change detection.
5. Syncing all stacks to Terramate Cloud.

Prerequisites

To start using Terramate, let's quickly run through some steps to ensure the correct setup of your environment.

Install Terramate

macOSUbuntu & DebianFedora & CentOSWindows

```
brew install terramate
```

Other installation options are available. When the installation is complete, you can test it out by reading the current version:

```
$ terramate version
```

Install Terraform or OpenTofu

Next, install Terraform or OpenTofu:

- Install Terraform
- Install OpenTofu

Create a new project

Terramate requires a git repository, and every git repository is a project in Terramate.

Let's create a new repository terramate-quickstart to set up your first Terramate project.

```
$ git init -b main terramate-quickstart

$ cd terramate-quickstart
```

New git repositories are empty per default and don't contain any commits. The change detection in Terramate works by detecting changes between at least two commits. Let's add an initial, empty commit to the repository:

```
$ git commit --allow-empty -m "Initial empty commit"
```

Create stacks

Now that the repository is ready, you can create your first stack. Stacks in Terramate are a collection of infrastructure resources that you configure, provision, and manage as a unit.

We will give the stack an ***optional*** name and description upon creation to track the purpose and details of a stack.

Terramate will ensure that on creation, each stack gets an id set automatically if not defined by the user.

```
$ terramate create \
  --name "Alice" \
  --description "Alice's first stack" \
  stacks/alice
```

The terramate create command creates a file stack.tm.hcl containing a stack {} block to configure the stack, which will look something like this.

```
$ cat stacks/alice/stack.tm.hcl


stack {
  name        = "Alice"
  description = "Alice's first stack"
  id          = "5b33e1c4-a3b0-477d-b0f1-add5918f764d"
}
```

How does Terramate detect stacks?

Stacks in Terramate are identified by a directory that includes a *.tm.hcl file, which contains a stack {} block. The file can have any name but the terramate create command always creates a file named stack.tm.hcl

Next, let's check in our newly created stack to the repository:

```
$ git add stacks/alice/stack.tm.hcl

$ git commit -m "Create a first stack with Terramate"
```

To verify that Terramate is aware of the new stack, you can run terramate list, which returns a list of all stacks available in your project.

```
$ terramate list

stacks/alice
```

To create a second stack, we follow the same commands. First, we create the stack:

```
$ terramate create \

  --name "Bob" \

  --description "Bob's first stack" \

  stacks/bob
```

Next, we add the second stack to our repository:

```
$ git add stacks/bob/stack.tm.hcl

$ git commit -m "Create a second stack with Terramate"
```

To verify that Terramate is aware of both stacks, we can run terramate list again.

```
$ terramate list

stacks/alice

stacks/bob
```

Change detection in action

Since we created our stacks step by step and created a git commit per stack, we can leverage change detection to see what changes we introduced

in our latest commit.

```
$ terramate list --changed

stacks/bob
```

By running the command mentioned above, you will see only the second stack is listed now, as we newly introduced the second stack without changing the first stack. Terramate's change detection is based on a Git integration but also supports more integrations like Terraform to detect affected stacks using a local Terraform Module that has been updated outside of the stack.

Code generation

Empty stacks are of not much use. One of Terramate's primary use cases is orchestrating IaC tools such as Terraform and generating code for it - but Terramate is not limited to Terraform and can also be used with other tooling such as OpenTofu, Terragrunt, Kubernetes, Helm, CloudFormation, etc.

Every Terraform stack will need a backend configuration. For the sake of this guide, we will use the Terraform local backend.

To generate backend code we create a file called stacks/backend.tm.hcl:

```
$ cat <<EOF >stacks/backend.tm.hcl

generate_hcl "backend.tf" {

  content {

    terraform {

      backend "local" {}

    }

  }

}
```

This configures Terramate to generate a backend.tf file in every stack it can reach within the stacks/ directory. In this case, it applies to our alice and bob stacks.

To trigger the code generation we need to run the terramate generate command:

```
$ terramate generate
```

```
Code generation report


Successes:


- /stacks/alice

  [+] backend.tf


- /stacks/bob

  [+] backend.tf


Hint: '+', '~' and '-' mean the file was created, changed and deleted,
respectively.
```

The generation report will report any changes in the generated code.

Let’s commit the changes and generated code:

```
$ git add stacks

$ git commit -m 'Add a backend configuration to all stacks'
```

Orchestration in action

The stacks created in the previous sections represent isolated environments, often referred to as "root modules" in Terraform and OpenTofu. To make them functional, we must run terraform init or tofu init in both. Terramate allows you to orchestrate any command in stacks using the `terramate run` command.

But before we can start, we need to prepare git to ignore temporary Terraform files by adding a .gitignore file, which is located in the root directory of our repository:

```
# NOTE:

# You might not want to add state and lock file here
```

```
# This is just convenient when running the quickstart guide
$ cat <<EOF >.gitignore
.terraform
.terraform.lock.hcl
*.tfstate
terraform.tfstate
terraform.tfstate.backup
*.tfplan
$ git add .gitignore
$ git commit -m 'Add .gitignore'
```

Now let’s initialize our stacks:

```
$ terramate run terraform init
```

And run a Terraform plan:

```
$ terramate run terraform plan
```

Add Terraform resources

In this section, we create a Terraform null resource for demonstration. Null Resources do not need to configure any cloud credentials as they do not create real resources but only virtual ones.

This example will show:

- You can use plain Terraform config in any stack without using code generation.
- Running only on changed stacks can save us time running and reviewing.

```
$ cat <<EOF >stacks/bob/null.tf
resource "null_resource" "quickstart" {
}
```

```
$ git add stacks/bob/null.tf

$ git commit -m "Add a null resource"
```

To apply the changes, re-initialize Terraform and run terraform apply in the updated stacks. As we only added the resource to the bob stack, we can leverage change detection to run in the changed stack only.

Running commands only in stacks containing changes allows us to keep execution run times fast and blast radius small.

Re-initialize Terraform to download the null provider:

```
$ terramate run --changed terraform init
```

Preview a plan:

```
terramate run --changed terraform plan
```

After reviewing the plan, we can apply the changes:

```
terramate run --changed terraform apply -auto-approve
```

When running the terraform plan again, we expect no changes to be planned anymore:

```
terramate run --changed terraform plan
```

Connecting Terramate Cloud

Now that you have created your first Terramate project, let's connect it to Terramate Cloud and learn how we can enable observability, CI/CD, drift detection, and more! Terramate CLI and Terramate Cloud work in tandem to deliver a reliable experience. Terramate Cloud is free for individual use, with features available for teams.

Create a cloud account

To start using the cloud, you need to sign up for a free cloud account and create an organization.

- Sign in and Sign up to cloud.terramate.io and
- Create your Organization

Remember the organization's short name that you set for accessing the organization on Terramate Cloud to configure your Terramate CLI in the next steps.

Configure your repository

Configure your Terramate project to sync data to your Terramate Cloud organization after creating it.

```
$ cat <<EOF >terramate.tm.hcl

terramate {

  config {

    cloud {

      organization = "organization-short-name" # TODO: fill in your org short name

    }

  }

}
```

```
$ git add terramate.tm.hcl

$ git commit -m "Add Terramate Cloud configuration"
```

Create a GitHub repository

Terramate Cloud requires a GitHub, GitLab, or BitBucket repository to work properly. We will use GitHub in this guide. Let's start by creating a new repository in your personal GitHub account or an organization. We'll name the repository terramate-quickstart and create it as a private repository.

Once the repository is created on GitHub, you can add it to your local repository and push your data to GitHub. Don't forget to replace your-account with your GitHub account or organization handle.

```
$ git remote add origin git@github.com:your-account/terramate-quickstart.git
```

```
$ git branch -M main

$ git push -u origin main
```

Login from CLI

To synchronize data from your local machine, you will need to login to Terramate Cloud from the CLI. Terramate CLI will store a session on your machine after a successful login.

Use the following command to initiate the login.

```
$ terramate cloud login
```

If you want to login with GitHub instead, use:

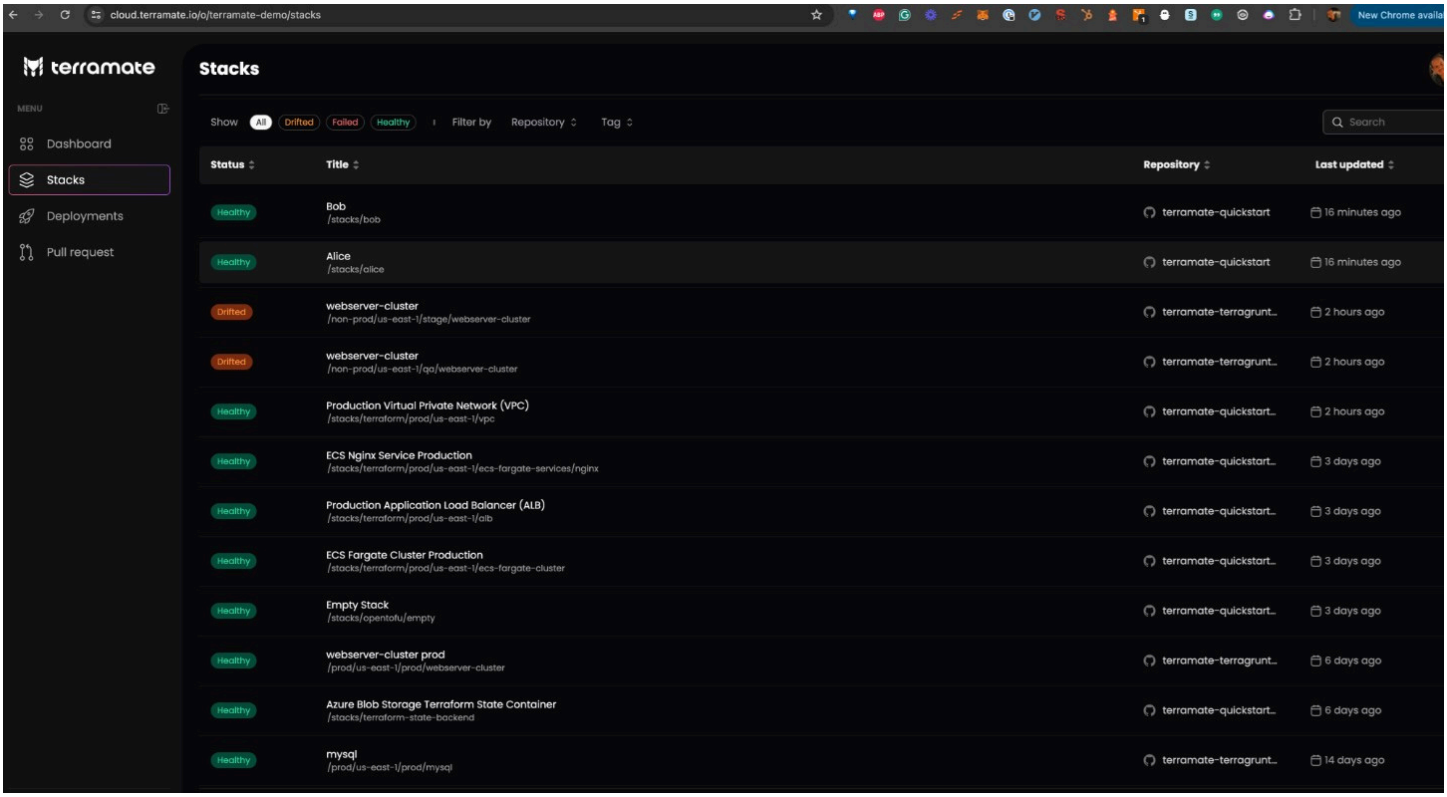
```
$ terramate cloud login --github
```

Sync stacks to Terramate Cloud

After setting up your GitHub repository and Terramate Cloud organization, let's sync the stacks configured in our Terramate project to Terramate Cloud. The easiest to sync your stacks is to run a drift detection workflow in all stacks and sync the result to Terramate Cloud:

```
$ terramate run \
  --sync-drift-status \
  --terraform-plan-file=drift.tfplan \
  --continue-on-error \
  -- \
  terraform plan -detailed-exitcode -out drift.tfplan
```

In a nutshell, the command above runs a terraform plan or tofu plan in all your stacks and sends the result to Terramate Cloud. Since the plans don't detect any changes, Terramate Cloud won't mark those stacks as drifted but only adds them to your inventory of stacks.



Trigger a deployment

As a final step of this guide, we will change one of our stacks and trigger a new deployment. For that, let's add another null resource to our bob stack.

```
$ cat <<EOF >>stacks/bob/null.tf

resource "null_resource" "quickstart2" {

}

EOF


$ git add stacks/bob/null.tf

$ git commit -m "Add another null resource"
```

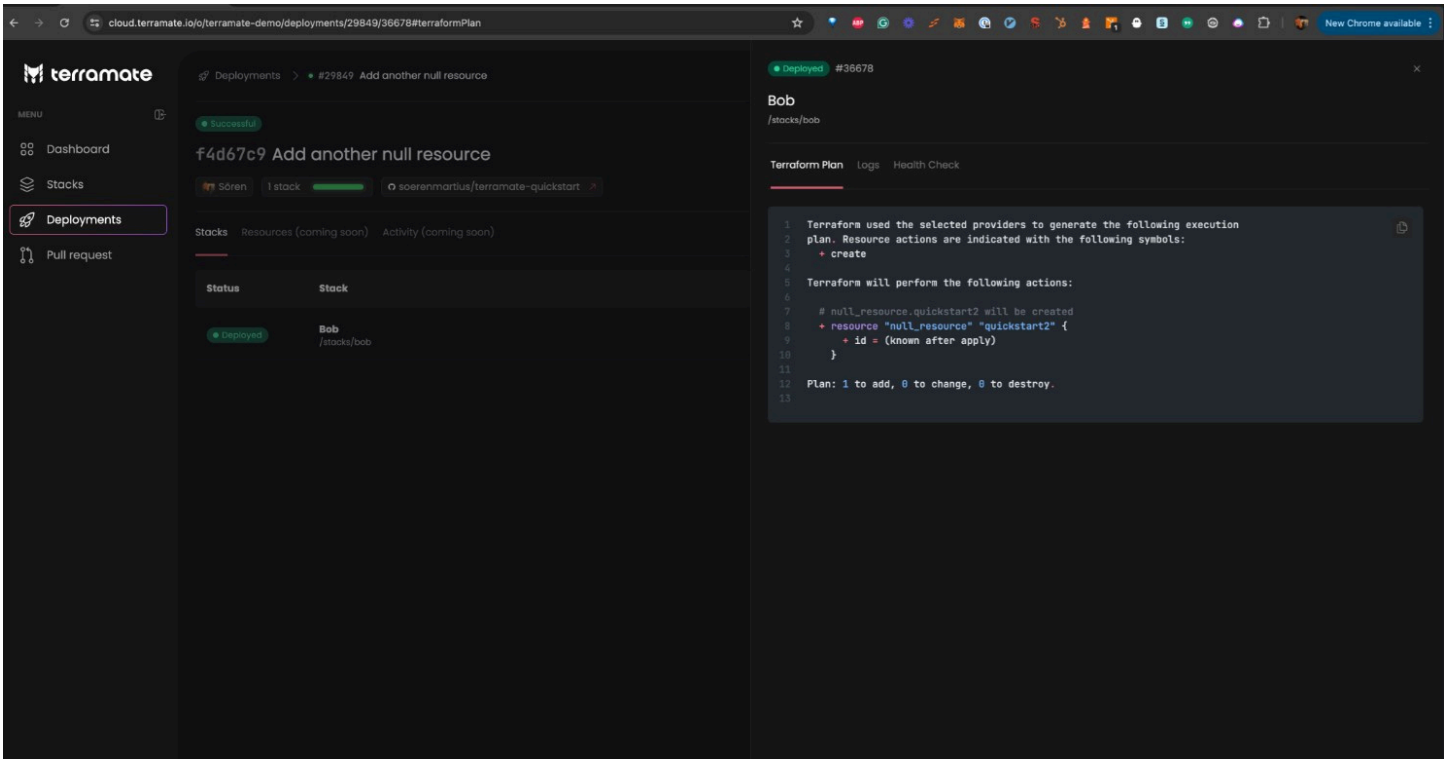
Next, plan and apply the changes. The following command will create a plan in all changed stacks and apply the generated plan files and sync the result as a deployment to Terramate Cloud.

```
$ terramate run \

  --changed \
```

```
  -- \
  terraform plan -lock-timeout=5m -out deploy.tfplan


$ terramate run \
  --changed \
  --sync-deployment \
  --terraform-plan-file=deploy.tfplan \
  -- \
  terraform apply -input=false -auto-approve -lock-timeout=5m deploy.tfplan
A new deployment has started in Terramate Cloud. Check out the Deployments section to see it.
```



EXAMPLE FOR A VM CODE

plaintext Copy code

```
terramate-vm/
├── main.tm.hcl
├── terramate.tm.hcl
├── stacks/
|   └── dev/
|       ├── main.tf
|       ├── variables.tf
|       ├── outputs.tf
|       └── terraform.tfvars
└── modules/
    └── vm/
        ├── main.tf
        ├── variables.tf
        └── outputs.tf
```

Terramate.tm.hcl

```
terramate {
  config {
    stacks {
      import {
        source = "./stacks"
      }
    }
  }
}
```

Stacks/dev/provider.tf

Just save the provider.tf code of terraform

```hcl
terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "3.98.0"
    }
  }
  #   backend "azurerm" {
  #   resource_group_name  = "ropra"
  #   storage_account_name = "roprastor"
  #   container_name       = "ropracontainer"
  #   key                  = "mod3.terraform.tfstate"
  # }
}

provider "azurerm" {
  features{}
}
```

Stacks/dev/variables.tf

Declared the variables

```hcl
variable "rggg" {

}
variable "vnett" {

}
variable "subnet" {

}
variable "vms" {
  type=map(any)
}
```

Stacks/dev/main.tf

Declared the main code

```
module "vm" {
  source              = "../../modules/vm"
  vm_name             = var.vm_name
  resource_group_name = var.resource_group_name
  location            = var.location
  admin_username      = var.admin_username
  admin_password      = var.admin_password
}
```

stacks/dev/terraform.tfvars

Declare the TFvars file as Inputs

stacks/dev/output.tf

```
output "vm_name" {
  value = module.vm.vm_name
}
```

module/vm/main.tf

```
resource "azurem_resource_group" "rg_vm" {
    for_each = var.rgs
    name=each.valvue.name
    location=each.value.location

}

resource "azurem_virtual_networks" "vnet_vm" {
    depends_on = [ var.rgs ]
  for_each = var.vnet
  name=each.value.name
  location=azurem_resource_group[].location
  resource_group_name=azurem_resource_group.name

}

```

module/vm/variable.tf

```
variable "rggg" {

}
variable "vnett" {

}
variable "subnet" {

}
variable "vms" {
  type=map(any)
}
```

module/vm/output.tf

```
output "vm_name" {
  value = azurerm_linux_virtual_machine.vm.name
}

```

NOW YOU CAN THIS CODE ON TERRAMATE CLI OR AS A SCRIPT ON THE TERRAFORM BUILD PIPELINE