Getting Started with Terraform

Module Overview

This module will cover the following:

- Infrastructure as Code Concepts
- Terraform Overview
- Common Terraform Tasks
- Rubrik & Terraform

IaC Concepts

What is Infrastructure as Code?

- A **practice** in which infrastructure automation is based on practices from software development.
- Emphasizes **consistent**, **repeatable** routines for **provisioning** and **modifying** systems and their configuration.
- Changes are made to **definitions** and then rolled out to systems through **unattended processes** that include thorough **validation**.

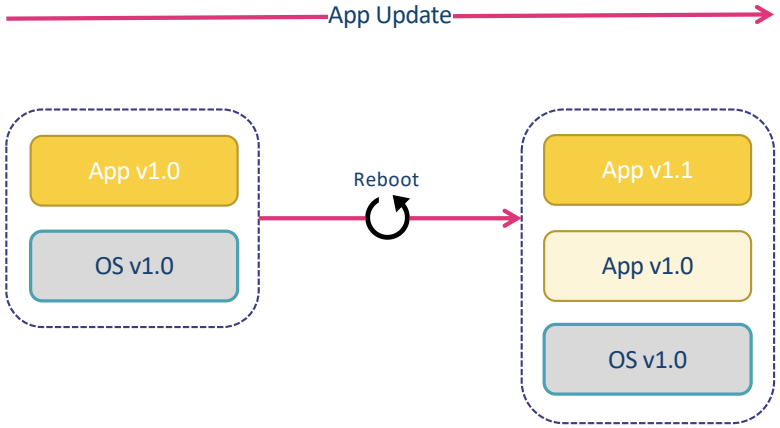
Declarative

- Terraform code is written in a declarative style, where you just define the **end state** needed for your infrastructure.

```
resource "ec2_instance" "example" {
 count = 5
 ami = "ami-afd15ed0"
 instance_type = "t2.micro"
}
```

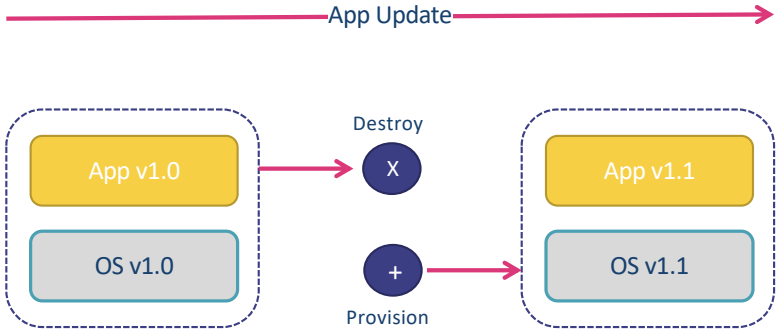
Mutable Infrastructure

- Infrastructure is continually updated, patched, and tuned to meet ongoing needs.
- CM tools, such as Chef or Puppet, typically default to mutable infrastructure paradigm.
- Over time, as more patches are applied, each server builds up a unique history of changes.
- As a result, each server is slightly different than the others, leading to configuration drift.



Immutable Infrastructure

- All changes result in deployment of new app version.
- Reduces likelihood of configuration drift.
- Automated testing is more effective; an immutable image that passes all tests is likely to behave the same in production.
- Blue / green



Introducing Terraform

Terraform

- Multi-platform
- Allows us to define what we need in a declarative manner
- Translates templates into API calls
- Saves current state as a file
- Detects diffs from current state to apply changes

```
> terraform apply
aws_instance.example: Refreshing state...
aws_instance.example: Modifying...
  tags.#:    "0" => "1"
  tags.Name: "" => "terraform-example"
aws_instance.example: Modifications complete

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

Terraform Providers

- **Providers** interact with APIs of the various IaaS/SaaS/PaaS services
- Each provider contains the many different resources that the service provides, such as networks, virtual servers, security groups, volumes, and so on

Common Terraform Tasks

Define a Provider (`main.tf`)

- Create a file called `main.tf`, this is where the provider and resources are defined
- This example makes use of the AWS provider to request EC2 instances in the region defined by a variable.

```
provider "aws" {
  region = "${var.aws_region}"
}
```

Define Variables (`variables.tf`)

- Create a file called `variables.tf`, this is where the provider and resources are defined
- This example makes use of a variable to define the AWS region

```
variable "aws_region" {
  description = "AWS region"
  default     = "eu-west-1"
}
```

Initialize Terraform

- Run `terraform init` in order to initialize various settings for Terraform that will create the required environment to proceed
- It will also download the necessary plugins for the selected provider

```
$ terraform init

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...
- Downloading plugin for provider "aws" (1.0.0)...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 1.0"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.
```

Specify Resources

- Define the desired outcome in the `main.tf`
- This example defines the creation of an instance based off the defined AMI, sized as `t1.micro`, and properly tagged

```
resource "aws_instance" "web" {
  ami           = "ami-70728c08"
  instance_type = "t2.micro"

  tags {
    Name = "HelloWorld"
  }
}
```

Execution Plan

- Generate the execution plan by running `terraform plan`
- Consider `terraform plan` as a dry-run of your changes

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  + aws_instance.web
      id:                               <computed>
      ami:                              "ami-70728c08"
      associate_public_ip_address:      <computed>
      source_dest_check:                "true"
      subnet_id:                        <computed>
      tags.%:                           "1"
      tags.Name:                        "HelloWorld"


Plan: 1 to add, 0 to change, 0 to destroy.
```

Provisioning

- Run `terraform apply` in order to have the changes applied
- This should be done once the execution plan has been review have reviewed and confirmed that changes are as expected

```
$ terraform apply
aws_instance.web: Creating...
  ami:                               "" => "ami-70728c08"
  instance_type:                     "" => "t2.micro"
  source_dest_check:                 "" => "true"
  subnet_id:                         "" => "<computed>"
  tags.%:                            "" => "1"
  tags.Name:                         "" => "HelloWorld"
  tenancy:                           "" => "<computed>"
  volume_tags.%:                     "" => "<computed>"
  vpc_security_group_ids.#:          "" => "<computed>"
aws_instance.web: Still creating... (10s elapsed)
aws_instance.web: Still creating... (20s elapsed)
aws_instance.web: Creation complete after 22s (ID: i-0c0b4a732311b8bce)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Removing Components

- To get rid of resources, use the `terraform destroy` command
- With this example, I could use a command to specifically name the resource to destroy:

  ```
  terraform destroy -
  target=aws_instance.web
  ```

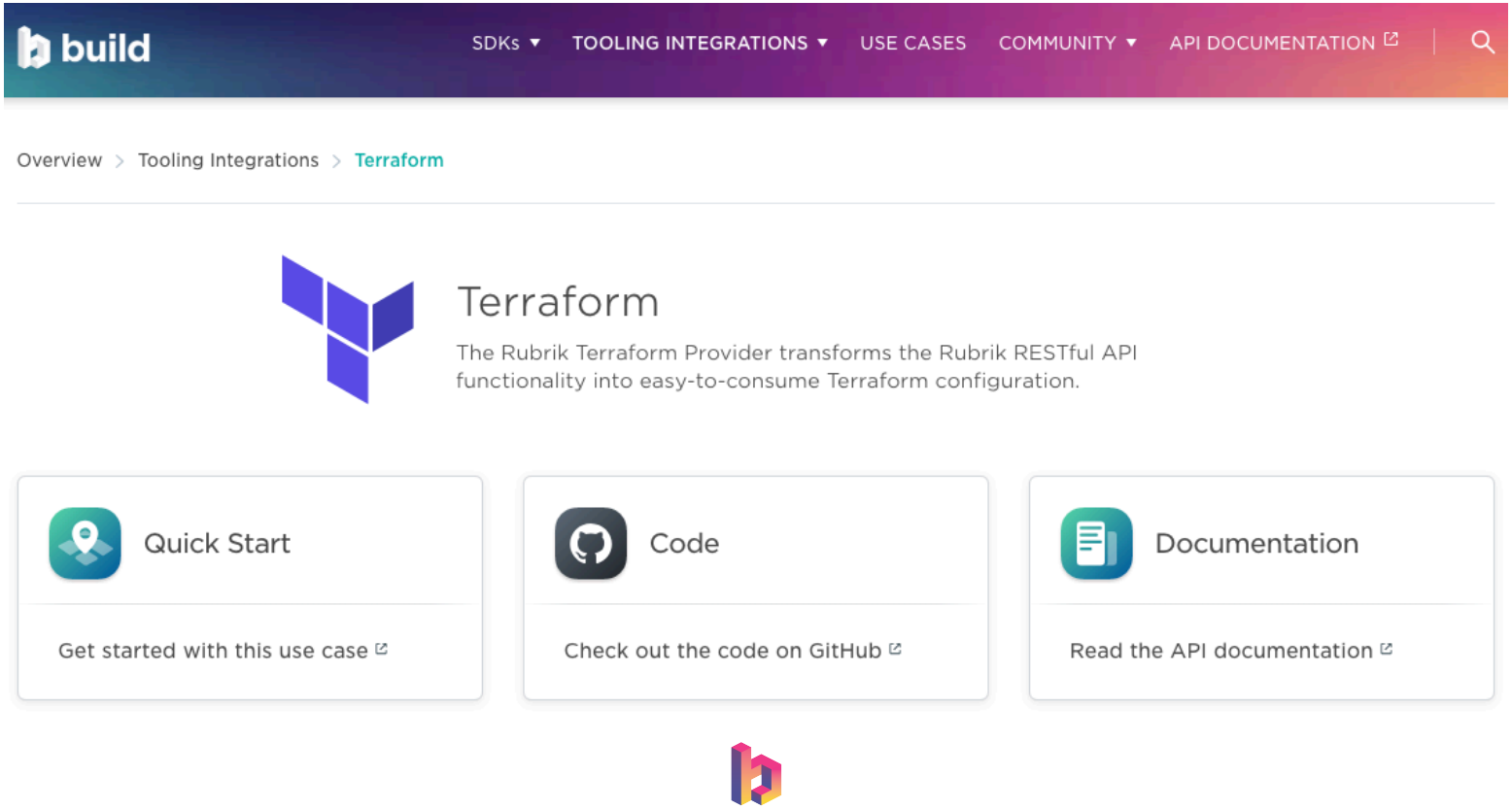
```
$ terraform destroy

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  - aws_elb.lb
  - aws_instance.web[0]
  - aws_instance.web[1]
  - aws_instance.web[2]

Plan: 0 to add, 0 to change, 4 to destroy.

Do you really want to destroy?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.
```

`tfstate` Files

- After an `apply` command is executed, the affiliated directory will contain two new files:
  - `terraform.tfstate`
  - `terraform.tfstate.backup`
- Configurations for resources provisioned using Terraform are stored so that when running `apply` again will compare the configuration to the local system version
- **Note:** any changes made manually to Terraform provisioned infrastructure will be overwritten by `terraform apply`

Rubrik & Terraform

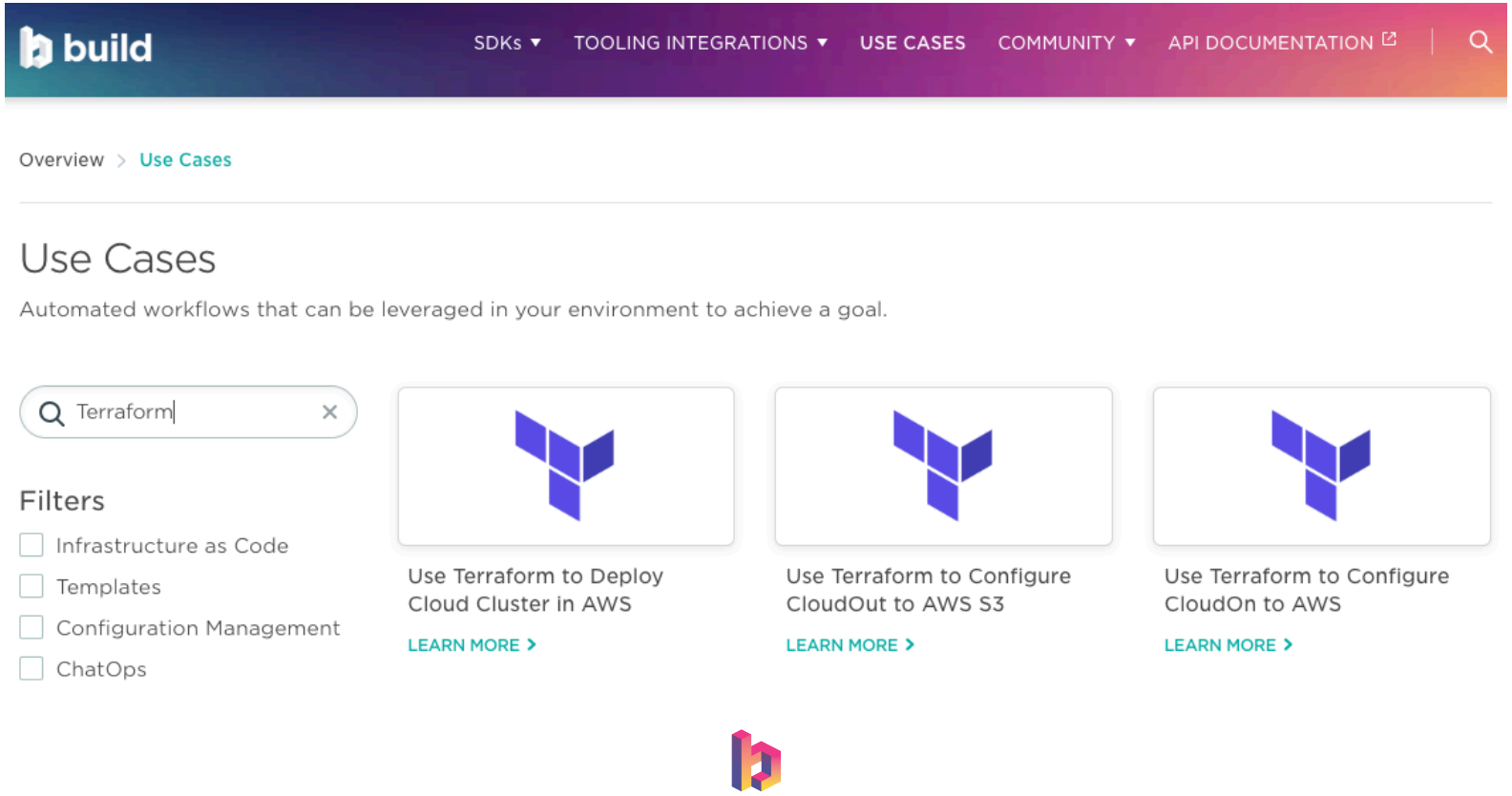
Rubrik Provider for Terraform



Modules

- Terraform modules are pre-packaged configuration files that can be used as building blocks to create new infrastructure items without writing code
- Modules are available publicly in the Terraform registry, and can be directly added to configuration files for quickly provisioning resources

Modules for Terraform




build
Building the Future of Data Management