Terraform 1.1 with AWS

Infrastructure as Code

Version: 1.2.0

Course Details 1/2

We have three days to cover the agenda.

- Module 01 - Overview of IaC
- Module 02 - Installation and Setup
- Module 03 - Virtual Private Cloud (VPC)
- Module 04 - Security Groups & EC2
- Module 05 - EC2 with Terraform
- Module 06 - From static to dynamic deployment
- Module 07 - Dependencies and Terraform
- Module 08 - S3 Storage and Terraform

Course Details 2/2

- Module 09 - Terraform an Modules
- Module 10 - Terraform and Elastic Load Balancing
- Module 11 - Terraform and Scaling
- Module 12 - Terraform and RDS
- Module 13 - Terraform and Serverless
- Module 14 - EKS with Terraform
- Module 15 - Remote State / Workspaces
- Module 16 - Terraform and Route53

Goals of the Course

- use Terrafrom to deploy Infrastructure in AWS
- getting a refresh on AWS services
- Hands-On Time

MOD 01 - of IaC

Manual Configuration Challenges

- Creating and configuring services is often done manually
- Documentation
- Reliability
- Reproducibility
  - Dev
  - Test
  - Prod

What is Infrastructure as Code?

> “Infrastructure as Code is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools”

Source: Wikipedia

Is Terraform the only one how does IaC?

Terraform – Template Example

This template creates a single EC2 instance in AWS

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

resource "aws_instance" "example-instance" {
  ami           = "ami-0e5b6b6a9f3db6db8"
  instance_type = "t2.micro"
}
```

Terraform – Key capabilities

- Terraform is a tool for provisioning infrastructure
- supports many providers (cloud agnostic)
- many resources for each provider
- define resources as code in terraform templates (HCL)

PRODUCTS & TECHNOLOGY

Announcing HashiCorp Terraform 1.0 General Availability

Terraform 1.0 — now generally available — marks a major milestone for interoperability, ease of upgrades, and maintenance for your automation workflows.

JUN 08 2021 | KYLE RUDDY

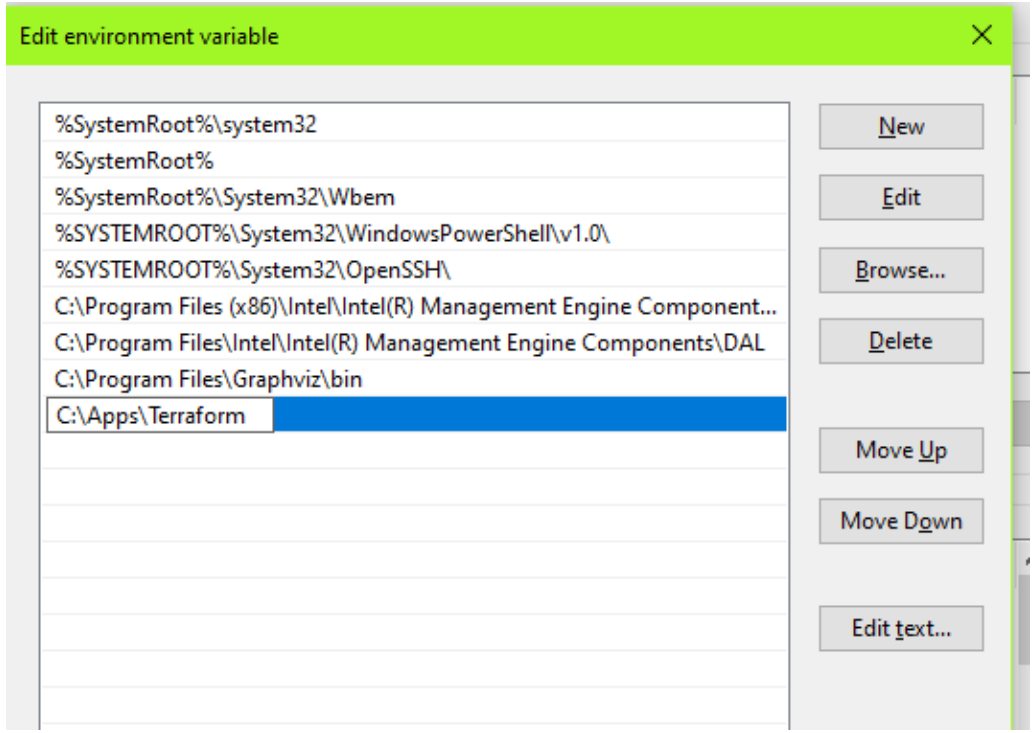
Terraform 1.0 - What are the benefits?

- Extended Maintenance Periods
  (1.x releases have 18 month maintenance period)
- More mature and stable
  (essentially a 0.15 super-service pack)
- Terraform state is cross-compatible between versions
  (0.14.x, 0.15.x, and 1.0.x.)

MOD 02 - Installation and Setup

Windows install

- Download the single binary from terraform-website
- move it to a `Directory`, for example `C:\Apps\Terraform`
- add it to your Systems-Path: `Control Panel` -> `System` -> `System settings` -> `Environment Variables`



MacOS/Linux install

- Download the single binary from terraform-website
- put it in /usr/local/bin

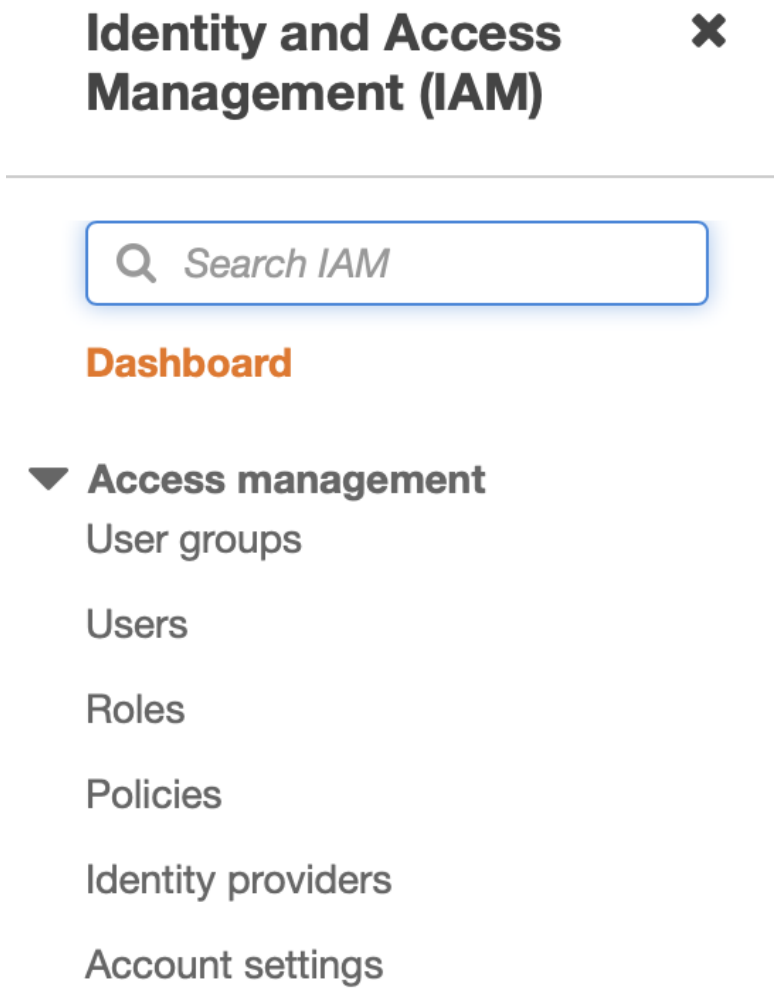
```
$ mv ~/Downloads/terraform /usr/local/bin/
```

- have fun

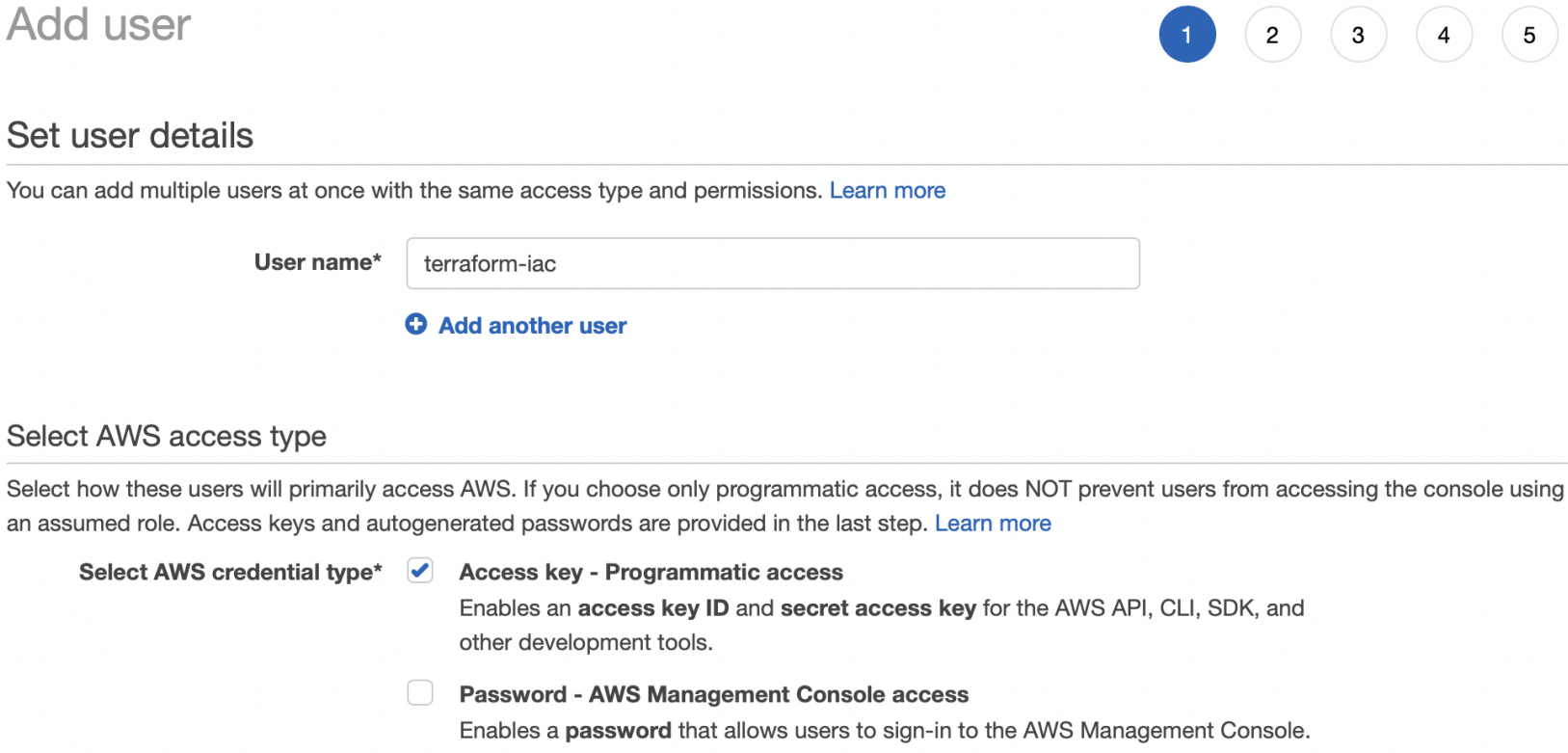
terraform-user setup

- login to AWS-Console
- switch to IAM-Service
- create a new user
- choose only
  Access key - Programmatic access
- add the administrator role
- note the access and secret key

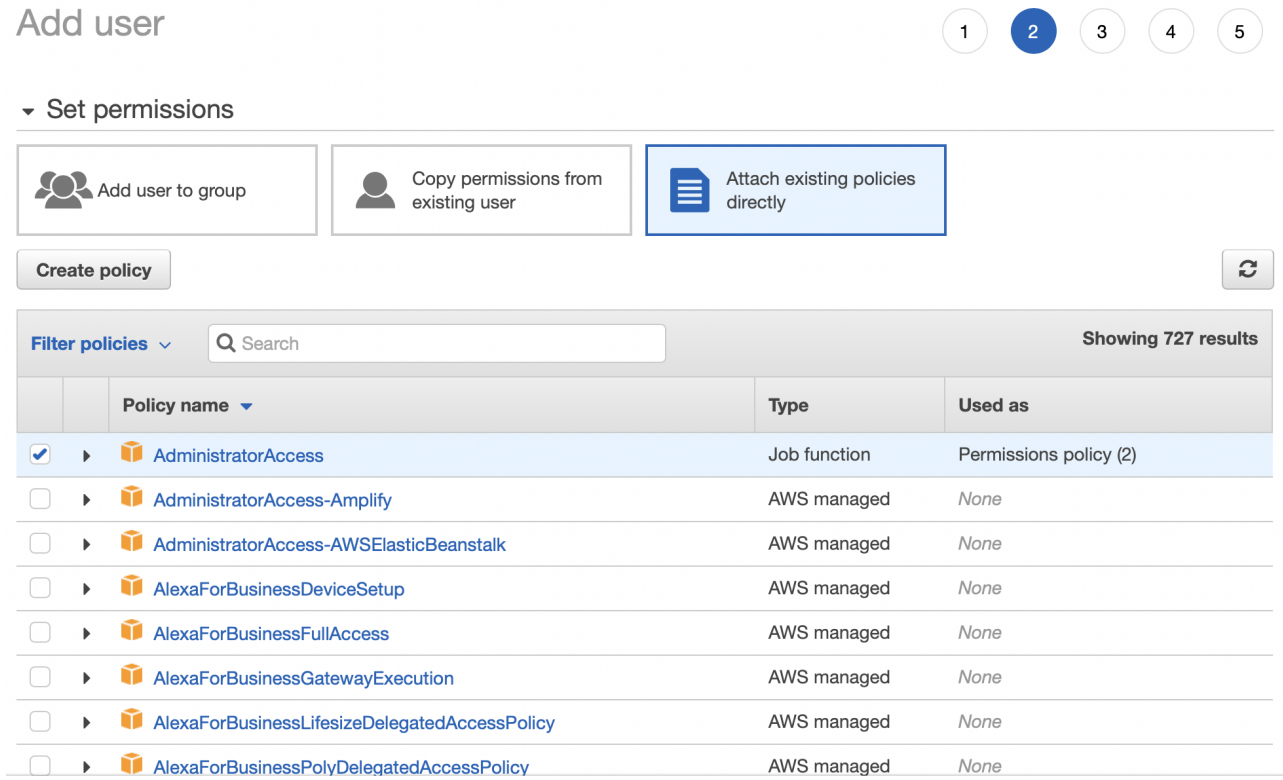
IAM Step 1/6


Identity and Access Management (IAM)
Search IAM
Dashboard
Access management
User groups
Users
Roles
Policies
Identity providers
Account settings

IAM Step 2/6


Add user
1
2
3
4
5
Set user details
You can add multiple users at once with the same access type and permissions. Learn more
User name*
terraform-iac
Add another user
Select AWS access type
Select how these users will primarily access AWS. If you choose only programmatic access, it does NOT prevent users from accessing the console using an assumed role. Access keys and autogenerated passwords are provided in the last step. Learn more
Select AWS credential type*
Access key - Programmatic access
Enables an access key ID and secret access key for the AWS API, CLI, SDK, and other development tools.
Password - AWS Management Console access
Enables a password that allows users to sign-in to the AWS Management Console.

IAM Step 3/6


Add user
1
2
3
4
5
Set permissions
Add user to group
Copy permissions from existing user
Attach existing policies directly
Create policy
Filter policies
Search
Showing 727 results
Policy name
Type
Used as
AdministratorAccess
Job function
Permissions policy (2)
AdministratorAccess-Amplify
AWS managed
None
AdministratorAccess-AWSElasticBeanstalk
AWS managed
None
AlexaForBusinessDeviceSetup
AWS managed
None
AlexaForBusinessFullAccess
AWS managed
None
AlexaForBusinessGatewayExecution
AWS managed
None
AlexaForBusinessLifesizeDelegatedAccessPolicy
AWS managed
None
AlexaForBusinessPolyDelegatedAccessPolicy
AWS managed
None

IAM Step 4/6

Add user

1 2 3 4 5

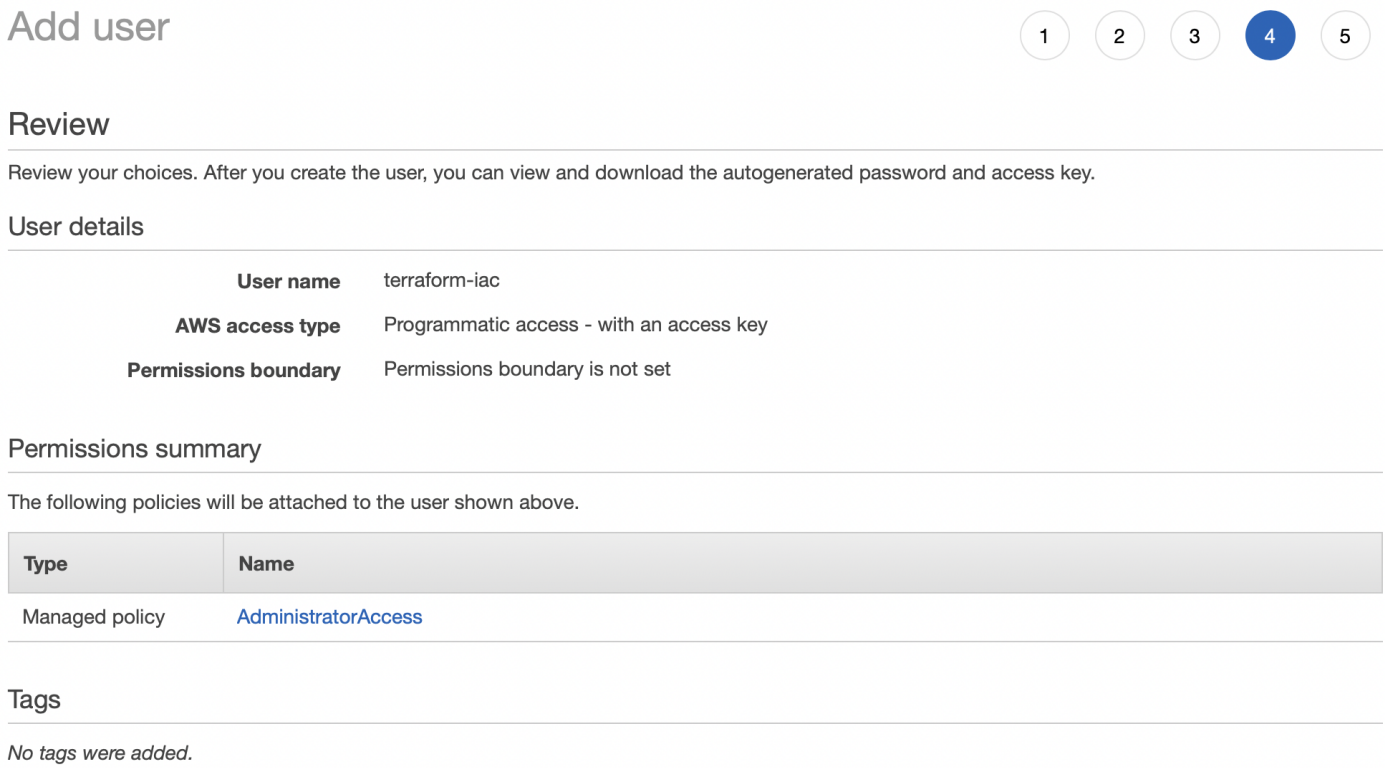
Add tags (optional)

IAM tags are key-value pairs you can add to your user. Tags can include user information, such as an email address, or can be descriptive, such as a job title. You can use the tags to organize, track, or control access for this user. Learn more

Key	Value (optional)	Remove
Add new key		

You can add 50 more tags.

IAM Step 5/6


Add user
1
2
3
4
5
Review
Review your choices. After you create the user, you can view and download the autogenerated password and access key.
User details
User name
terraform-iac
AWS access type
Programmatic access - with an access key
Permissions boundary
Permissions boundary is not set
Permissions summary
The following policies will be attached to the user shown above.
Type
Name
Managed policy
AdministratorAccess
Tags
No tags were added.

IAM Step 6/6

Add user

1 2 3 4 5

Success

You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.

Users with AWS Management Console access can sign-in at: https://833515639939.signin.aws.amazon.com/console

Download .csv

		User	Access key ID	Secret access key
▸	✓	terraform-iac	AKIA4EELHKSBRGHOIW6F	********* Show

Export the AWS-Credentials

set the value (Linux/MacOS)

```
export AWS_ACCESS_KEY_ID="AKIA4EELHKSBY6NZMA76"
export AWS_SECRET_ACCESS_KEY="dtnRM/6+O"
```

set the value (PowerShell/Windows)

```
$env:AWS_ACCESS_KEY_ID="AKIA4EELHKSBY6NZMA76"
$env:AWS_SECRET_ACCESS_KEY="dtnRM/6+O"
```

simple ec2-instance example

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

resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"

  tags = {
    Name = "ExampleAppServerInstance"
  }
}
```

Question: Why is this a weak example in the sense of IaC and **not** AWS perspective?

Terraform Version constraints

specify a range of acceptable versions (">= 1.2.0, < 2.0.0")

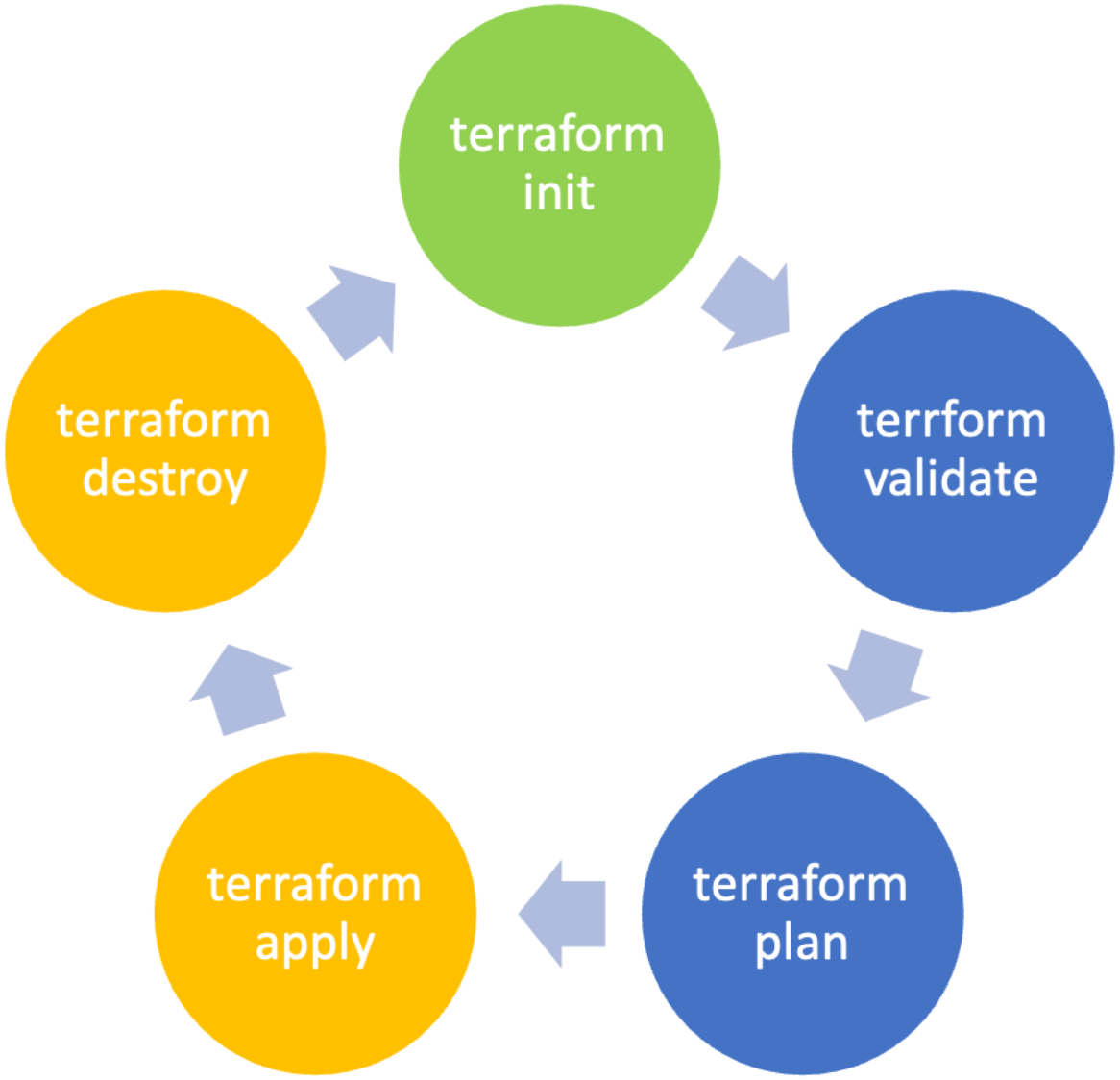
for this course we use provider aws 3.72.X, even if the 4.x branch is active now.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">=3.74"
    }
  }
  required_version = ">=1.0"
}
```

a better approach for a simple ec2-instance

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.74"
      }
  }
  required_version = ">= 1.0"
}

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

resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"
  tags          = {
    Name = "ExampleAppServerInstance"
  }
}
```

Terraform – Core Loop



LAB

Setup & "Hello Infra"

- Install and Setup Terraform
- create IAM User in AWS (AWS-CLI/Console)
- Initialize the aws-Provider
- define EC2-Instance and apply

Working with a state file

- Terraform saves everything about the instance to special file(s)
- Same directory as template file with `.tfstate` extension
  - terraform.tfstate
  - terraform.tfstate.backup
- The statefile should **not** be **committed** into version control
- This can be a problem on multi-developer env´s (more about that tomorrow)

Create a .gitignore for state-files

```
# Local .terraform directories
**/.terraform/*

# .tfstate files
*.tfstate
*.tfstate.*

# Crash log files
crash.log

# Exclude all .tfvars files, which are likely to contain sentitive data
*.tfvars
```

Source: GitHub

LAB

manage drift with Terraform

- check `terraform.tfstate`
- change Instance-Type to `t3.micro`
- add a `costcenter=42` Tag
- `apply` changes
- check the statefile again
- change costcenter via Dashboard
- `terraform plan` and check if TF can manage this drift

Terraform *Standard* Filelayout

File / Folder	Purpose
main.tf	Terraform Config and Constraints
outputs.tf	Output like IPs, Addresses, etc
providers.tf	Provider-Specific (Cred.)
resources.tf	for small projects
variables.tf	place for specifying variables
README.md	Documentation
env	folder place for tfvar-files

LAB

please refactor your code

- add `providers.tf` and refactor
- add `ressources.tf` and refactor
- create empty file `variables.tf`
- create empty file `outputs.tf`
- create an `env` folder

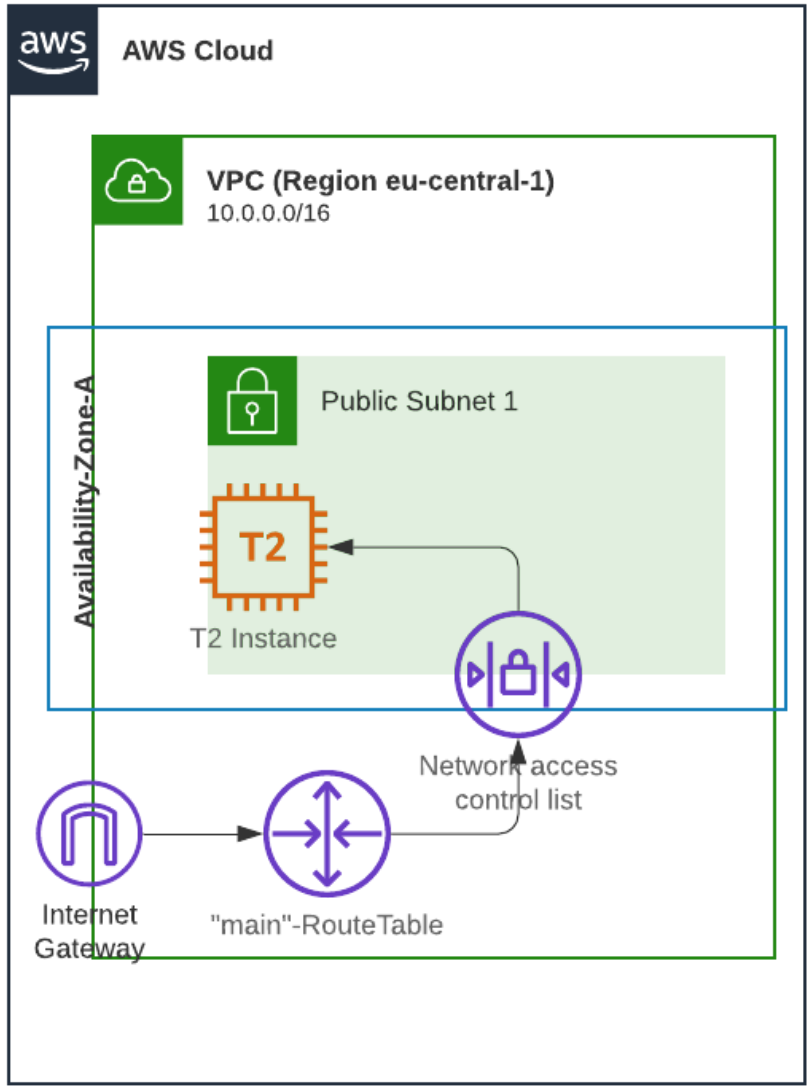
MOD 03 - Virtual Private Cloud (VPC)

Virtual Private Cloud

- A VPC is a virtual network dedicated to your AWS account
- Requires an IPv4 address space and optionally IPv6 address ranges
- Enables you to create specific CIDR ranges for your resources to occupy
- Provides strict access rules for inbound and outbound traffic.

Components of a basic VPC

- VPC CIDR Block
- public Subnet
- Internet-Gateway (or NAT)
- "main"-Route Table
- Network Access Control List



Ressource aws_vpc

Amazon Virtual Private Cloud (Amazon VPC) enables you to launch AWS resources into a virtual network that you've defined.

```
resource "aws_vpc" "my_vpc" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_hostnames = true

  tags = {
    Name = "My VPC"
  }
}
```

Ressource aws_subnet

A subnet is a range of IP addresses in your VPC. After creating a VPC, you can add one or more subnets in each Availability Zone.

```
resource "aws_subnet" "public" {
  vpc_id            = aws_vpc.my_vpc.id
  cidr_block        = "10.0.0.0/24"
  availability_zone = "eu-central-1a"

  tags = {
    Name = "Public Subnet"
  }
}
```

Ressource aws_internet_gateway

An internet gateway is a horizontally scaled, redundant, and highly available VPC component that enables communication between your VPC and the internet.

```
resource "aws_internet_gateway" "my_vpc_igw" {
  vpc_id = aws_vpc.my_vpc.id

  tags = {
    Name = "My VPC - Internet Gateway"
  }
}
```

Ressource aws_route_table

A route table contains a set of rules, called routes, that are used to determine where network traffic from your subnet or gateway is directed.

```
resource "aws_route_table" "my_vpc_eu_central_1a_public" {
  vpc_id = aws_vpc.my_vpc.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.my_vpc_igw.id
  }

  tags = {
    Name = "Public Subnet Route Table."
  }
}
```

Ressource aws_route_table_association

```
resource "aws_route_table_association" "my_vpc_eu_central_1a_public" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.my_vpc_eu_central_1a_public.id
}
```

LAB

create a basic VPC

- create a VPC in `eu-central-1`
- create one `public Subnet` for one Availability Zone
- create an `Internet-GW`
- create a `Route Table`
- associate the Route-Table with a Subnet

MOD 04 - Security Groups & EC2

Security Groups

- acts as a virtual firewall
- controls your EC2 instances inbound and outbound traffic
- act at the instance level, not the subnet level
- assign up to five security groups to an instance
- by default, they allow all outbound traffic

Create Security Groups- and Rule-Objects

```
resource "aws_security_group" "web_access" {
  name        = "web_access"
  description = "Allow port 80 access from outside world"
}

resource "aws_security_group_rule" "allow_webserver_access" {
  type              = "ingress"
  from_port         = 80
  to_port           = 80
  protocol          = "tcp"
  cidr_blocks       = ["0.0.0.0/0"]
  security_group_id = aws_security_group.web_access.id
}
```

Use Security Groups with blocks

```
resource "aws_security_group" "ssh_access" {
  name        = "web_security_group"
  description = "Terraform web security group"
  vpc_id      = "vpc-47111266642"

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

LAB

Security Groups

- create a security-group "webserver-access"
- add ingress-rule for port 22 and port 80 open to the world
- add egress-rule for everything open to the world (if necessary)
- attache it to app_server-ressource

MOD 05 - EC2 with Terraform

Attach a Security-Group to EC2

Security Groups including Roles needs to be attached to EC2-Instance for use. You can easily achive this in Terraform with,

- a reference
  `vpc_security_group_ids = [aws_security_group.allow_ssh.id]`
- or use of the ID
  `vpc_security_group_ids = "sg-4711"`

Deploy an EC2 instance to a VPC/Subnet

if you are don´t want the `default` -VPC as the EC2 target, you need to specifiy a Subnet-Id from the VPC of your choice in the `aws_instance` -Ressource

```
subnet_id = "subnet-0c58bb979af8269a7"
```

This will handle the deployment to the associated VPC of the Subnet.

SSH-Key-Pair

A key pair, consisting of a public key and a private key, is a set of security credentials that you use to prove your identity when connecting to an Amazon EC2 instance. Amazon EC2 stores the public key on your instance, and you store the private key

```
resource "aws_key_pair" "my-pub-key" {
  key_name   = "aws-pub-key"
  public_key = "ABCDEFGXXXXXXXXXXXX"
}
```

for aws_instance usage

reference it `key_name = aws_key_pair.my-pub-key.key_name`

or use the name `key_name = "aws-pub-key"`

create a keypair

This template creates a new keypair and download the `*.pem` File.

```
resource "tls_private_key" "pk" {
  algorithm = "RSA"
  rsa_bits  = 4096
}

resource "aws_key_pair" "kp" {
  key_name   = "myDemoKey" # Create a "myDemoKey" to AWS!!
  public_key = tls_private_key.pk.public_key_openssh
}

resource "local_file" "ssh_key" {
  filename = "${aws_key_pair.kp.key_name}.pem"
  content  = tls_private_key.pk.private_key_pem
}
```

Bootstrap with user_data

Usually we can use SSH-Access to install software manually or use something like Ansible.

Here we will use the clout-init-hook `user_data` from an EC2-Ressource.

```
user_data = << EOF
  #!/bin/bash
  sudo apt-get update
  sudo apt-get install -y apache2
  sudo systemctl start apache2
  sudo systemctl enable apache2
  echo "<h1>Deployed via Terraform</h1>" > /var/www/html/index.html
EOF
```

getting replicas with `count` and simple expressions

This template creates 4 single EC2 instances in AWS

```
resource "aws_instance" "app_server" {
  count         = 4
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"

  tags = {
    Name = "App Server ${count.index+1}"
  }
}
```

Cloudwatch - basics

- Cloudwatch is the AWS Monitoring Service
- you can create alarms for metrics
- The level of CloudWatch monitoring could be
  - basic (5 Min./Datapoint)
  - detailed (1 Min./Datapoint)
- Instances will be monitored by default in basic-level
- to enable detailed monitoring use

  `monitoring = true`

Cloudwatch Alarms

```
resource "aws_cloudwatch_metric_alarm" "alarm-foobar" {
  count                     = 3
  alarm_name                = "SRV-Alarm-${count.index}"
  comparison_operator       = "GreaterThanOrEqualToThreshold"
  evaluation_periods        = "2"
  metric_name               = "CPUUtilization"
  namespace                 = "AWS/EC2"
  period                    = "120"
  statistic                 = "Average"
  threshold                 = "80"
  alarm_description         = "This metric monitors ec2 cpu utilization"
  insufficient_data_actions = []

  dimensions = {
    "InstanceId" = aws_instance.app_server[count.index].id
  }
}
```

LAB

more attributes

create an EC2-Instance

- add a subnet-id from your VPC
- enable detailed monitoring
- deploy a basic webserver
- create two instances

MOD 06 - from static to dynamic deployment

Variables - simple types

There are simple types of variables you can set:

- string
- number
- bool

define a Variable (variables.tf)

```
variable "app_server_instance_type" {
  type         = string
  default      = "t2.micro"
  description = "The aws instance-type"
}
```

use a Variable (main.tf)

```
resource "aws_instance" "app_server" {
  count          = var.app_server_count
  ami            = var.ami_id
  instance_type = var.app_server_instance_type

  tags = {
    Name = "App Server ${count.index+1}"
  }
}
```

Custom validation with Rules I (variables.tf)

using a validation block nested within the variable block

```
variable "image-id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    condition     = length(var.image_id) > 4 && substr(var.image_id, 0, 4) == "ami-"
    error_message = "Image-id value must be a valid ami id, does it start with 'ami-'?"
  }
}
```

Custom validation with Rules II (variables.tf)

you can even use regex for this

```
variable "image-id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    # regex(...) fails if it cannot find a match
    condition     = can(regex("^ami-", var.image_id))
    error_message = "Image-id value must be a valid ami id, does it start with 'ami-'?"
  }
}
```

LAB

use Variables & Functions I

refactor the solution with

- variable `node_count`
- variable `ami_id`
- varible `instance_type`
- 3 replicas of your EC2 per default

Variables - Type Map

A Map is a lookup table, where you specify multiple keys with different values

```
# define a map of images
variable "images" {
  type = map(string)

  default = {
    eu-central-1 = "image-1234"
    us-west-1    = "image-4567"
  }
}

# getting the value for region eu-central-1
image_id = var.images["eu-central-1"]

# getting the correct value via a lookup
image_id = lookup(var.images, var.region)
```

Variables - Type List

A list value is an ordered sequence of strings indexed by integers starting with zero.

```
# define a map of images
variable "user_names" {
  type = "list(string)"
  default = ["Admin", "Jane", "Dane"]
}

# getting the value for the first entry
user = var.user_names[0]

# loop through in a ressource
count = length(var.user_names)
user  = var.user_names[count.index]
```

LAB

use Variables & Functions II

refactor the solution with

- variable `region`, default to **eu-central-1**
- change variable type `ami_id` to `map`

Setting Values with tfvars-Files

place Terraform variables in a special file (`*.tfvars`) and load them with the Terraform command:

```
foo = "bar"

somelist = ["one","two"]

somemap = {
  foo = "bar"
  bax = "qux"
}
```

Linux: `$ terraform apply -var-file=env/development.tfvars`

Windows: `$ terraform apply --var-file=env/development.tfvars`

LAB

use Variable-Files

refactor the solution with

- create `production.tfvars`
  - `region` to 'eu-central-1'
  - `instance_type` to t3.micro
  - `node_count` to 2
- create `development.tfvars`
  - `region` to 'eu-west-1'
  - `instance_type` to t2.micro
  - `node_count` to 1

use of environment variables

you can also supply values to your variables by using environment variables

Terraform will automatically read all environment variables with the `TF_VAR_` prefix

Example variables.tf

```
variable db_password {
  type = string
}
```

set the value (Linux)

```
export TF_VAR_db_password=Secret123
```

set the value (PowerShell)

```
$env:TF_VAR_db_password=Secret123
```

use Variables on the Commandline

To specify individual variables on the command line, use the `-var` option when running the terraform plan and terraform apply commands:

```
$ terraform apply -var="db_engine"="mysql"

$ terraform apply -var='user_names_list=["Peter","Paul","Marry"]

$ terraform apply -var='image_id_map={"us-east-1":"ami-abc123","us-east-2":"ami-def456"}
```

Variable Definition Precedence

Terraform loads variables in the following order (later sources taking precedence over earlier ones):

1. Environment variables
2. The terraform.tfvars file, if present.
3. The terraform.tfvars.json file, if present.
4. Any _.auto.tfvars or _.auto.tfvars.json files, processed in lexical order of their filenames.
5. Any -var and -var-file options on the command line, in the order they are provided. (This includes variables set by a Terraform Cloud workspace.)

Conditional Expressions

A conditional expression uses the value of a bool expression to select one of two values.

The syntax of a conditional expression is as follows:

```
condition ? true_val : false_val
```

Example with a region-default value if not set:

```
var.region != "" ? var.region : "eu-central-1"
```

LAB

use conditionals

- create a variable `var.server_build`
- create a conditional-expression if server should be build

Declaring an Output Value

Each output block that will be declared is exported after each `apply`:

What is the difference and when we use a) or b):

a)

```
output "app_server_ip_addr" {
  value = aws_instance.app_server.*.public_ip
}
```

b)

```
output "app_server_ip_addr" {
  value = aws_instance.app_server.public_ip
}
```

Terraform output on the commandline

You can use `terraform output` to get the latest info **from** the **state-file**

Output everything variable:

```
$ terraform output
app_server_public_ip = [
  "18.192.194.218",
]
```

output as JSON Object with jq-parsing

```
$ terraform output -json app_server_public_ip | jq -r '.[0]'
```

Sensitive Variables

Setting a variable as `sensitive` prevents Terraform from showing its value in the plan or apply output.

```
variable admin_password {
  type      = string
  sensitive = true
}
```

```
$ terraform apply -var=admin_password="Geheim123"
```

Sensitive Data and Output-Variables

try the difference of using sensitive on the output definition

```
output "db_admin_password" {
  description = "Admin Password"
  value       = var.admin_password
}
```

```
output "db_admin_password" {
  description = "Admin Password"
  value       = var.admin_password
  sensitive   = true
}
```

sensitive is NOT enough

Terraform will still record sensitive values in the **statefile**, and so anyone who can access the state data will have access to the sensitive values in cleartext.

```
$ grep -2 "password" terraform.tfstate
```

use Tools like:
Vault / AWS Secrets Mgmt. / Mozilla SOPS

more functions and dynamic blocks

using a more then one Tag for each Instance

```
variable "common_tags" {
  type = map(string)
  default = {
    Department  = "Global Infrastructure Services"
    Team        = "EMEA Delivery"
    CostCenter  = "12345"
    Application = "Intranet-Portal"
  }
}
```

use of a map with common tags

use it just as a variable:

```
resource "aws_instance" "app_server" {
  ...omitted output...
  tags = var.common_tags
  ...omitted output...
}
```

use it with the merge()-function

```
  tags = merge(var.common_tags, {
    Name = "AppSrv-${count.index + 1}"
    }
  )
```

LAB

use some common_tags

- create a map-variable
  `var.common_tags`
- set the tags:
  CostCenter = "12345"
  DeployedBy = "Terraform"
  SLA = "High"

file()-Function

we can use the file-function to dynamically load **local** files during deployment:

Example install_webserver.sh

```
#!/bin/bash
yum install httpd -y
/sbin/chkconfig --levels 235 httpd on
service httpd start
instanceId=$(curl http://169.254.169.254/latest/meta-data/instance-id)
region=$(curl http://169.254.169.254/latest/meta-data/placement/region)
echo "<h1>$instanceId from $region</h1>" > /var/www/html/index.html
```

Example main.tf in ressource ec2-instance

```
user_data = file("install_webserver.sh")
```

LAB

install a webserver at startup

refactor the solution

- create an install script for Apache
- add a user-data section to ec2
- use file-function to load the script
- check the webserver on the host

  `curl localhost`

dynamic Blocks in Terraform

Within top-level block constructs like resources, expressions can usually be used only when assigning a value to an argument using the `name = expression` form. This covers many uses, but some resource types include repeatable
nested blocks in their arguments, which typically represent separate objects that are related to (or embedded within) the containing object:

Create Security Groups with dynamic blocks (simple)

```
variable "ports" {
  default = [80, 443, 22]
}

resource "aws_security_group" "dynamic-demo" {
  name        = "demo-sg-dynamic"
  description = "Dynamic Blocks for Ingress"

  dynamic "ingress" {
    for_each = var.ports
    content {
      description = "description ${ingress.key}"
      from_port   = ingress.value
      to_port     = ingress.value
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}
```

Create Security Groups with dynamic blocks (with maps)

```
variable "sg_config" {
  type = map(any)
  default = {
    "web access" = {
      port = 80,
      cidr_blocks = ["0.0.0.0/0"],
    }
    "ssh access" = {
      port = 22,
      cidr_blocks = ["10.0.0.0/16"],
    }
  }
}
resource "aws_security_group" "map" {
  name        = "demo-map"
  description = "demo-map"

  dynamic "ingress" {
    for_each = var.sg_config
    content {
      description = ingress.key # IE: "description 0"
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = "tcp"
      cidr_blocks = ingress.value.cidr_blocks
    }
  }
}
```

LAB

using dynamic blocks:

- refactor the security-group "webserver-access" to use dynamic blocks
- add the following ingress-rules

Port	CIDR-Block	Description
22	only within VPC	ssh access
80	open to the world	web access
443	open to the world	tls web access

Datasources

Data sources allow Terraform use information defined outside of Terraform.

Examples: VPC-ID, AMI-ID, KeyPair, Hosted Zone Info, Textfiles, etc.

This is defined by another separate Terraform configuration, or modified by functions. Each provider may offer data sources alongside its set of resource types.

Datasource aws_ami

Use this data source to get the ID of a registered AMI for use in other resources.

```
data "aws_ami" "amazon-linux-2" {
  owners      = ["amazon"]
  most_recent = true

  filter {
    name   = "name"
    values = ["amzn2-ami-hvm-*-x86_64-gp2"]
  }
}
```

Datasource aws_vpc

`aws_vpc` provides details about a specific VPC.

This resource can prove useful when a module accepts a vpc id as an input variable and needs to, for example, determine the CIDR block of that VPC.

```
data "aws_vpc" "selected" {
  tags = {
    Owner = "Terraform"
    Name  = "terraform-example-vpc"
  }
}
```

Datasource aws_subnets_ids

`aws_subnet_ids` provides a set of ids for a vpc_id

This resource can be useful for getting back a set of subnet ids for a vpc.

```
data "aws_subnet_ids" "selected" {
  vpc_id = data.aws_vpc.selected.id
}
```

The following example retrieves a set of all subnets in a VPC with a custom tag of Tier set to a value of "Private".

```
data "aws_subnet_ids" "private" {
  vpc_id = var.vpc_id
  tags = {
    Tier = "Private"
  }
}
```

LAB

use a datasource

refactor the solution

- create a datasource to query the newest Amazon Linux 2 AMI
- create an ec2-instance and use the AMI dynamically

Datasource template_file 1/2

The template_file data source renders a template from a template string, which is usually loaded from an external file.

```
data "template_file" "userdata" {
  template = file("${path.module}/userdata.sh")
  vars = {
    DOCKER_VERSION  = var.DOCKER_VERSION
    ANSIBLE_VERSION = var.ANSIBLE_VERSION
  }
}
```

Datasource template_file 2/2

File userdata.sh

```
#!/bin/bash -xe
yum update -y

amazon-linux-extras install docker=${DOCKER_VERSION}  -y
amazon-linux-extras install ansible2=${ANSIBLE_VERSION} -y
yum install -y zip unzip
```

use the datasource in ec2.tf

```
resource "aws_instance" "webserver" {
  user_data  = data.template_file.userdata.rendered
  ...
}
```

MOD 07 - Dependencies and Terraform

Dependency Graphs in Terraform

- Terraform uses dependency graphs to determine the build and deletion order of resources
- Three different types of nodes in a Terraform graph:
  - Resource node
  - Provider configuration node
  - Resource meta-node

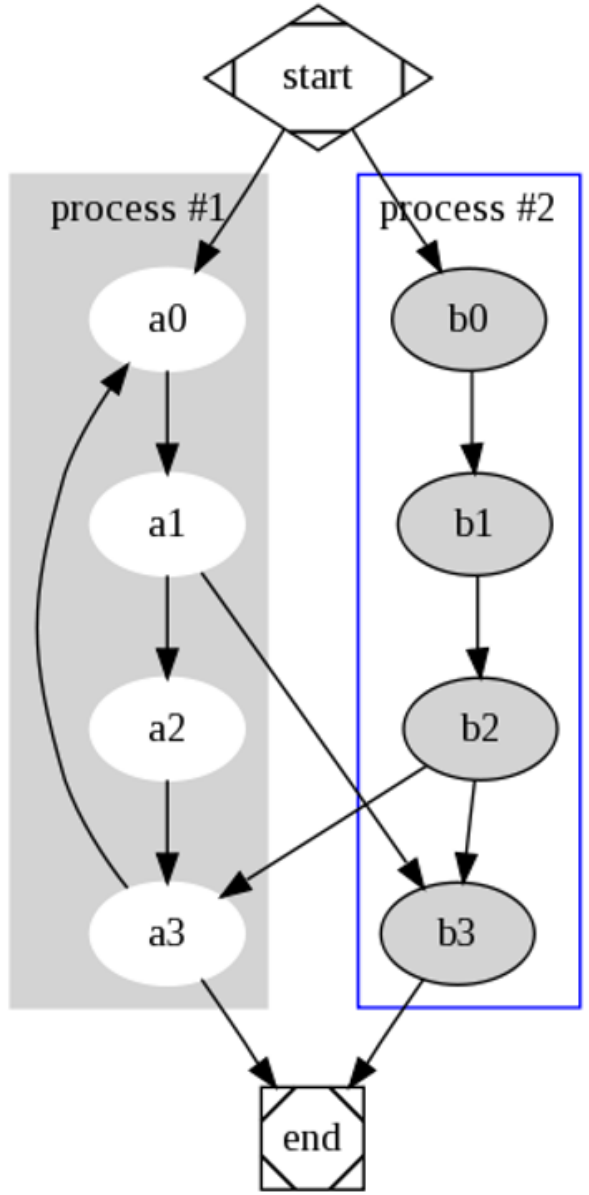
Controlling dependencies with depends_on

- Terraform will resolve dependencies automatically
- Sometimes built-in dependency resolution leads to to unwanted behavior
- Enforce dependencies: `depends_on`
  - Accepts a list of resources that this resource depends on
  - Resource won't be created until the ones listed inside this parameter are created

```
resource "aws_instance" "app_server" {
  ami              = var.ami_id
  instance_type    = var.instance_type

  depends_on = [
    aws_instance.db_server
  ]
}
```

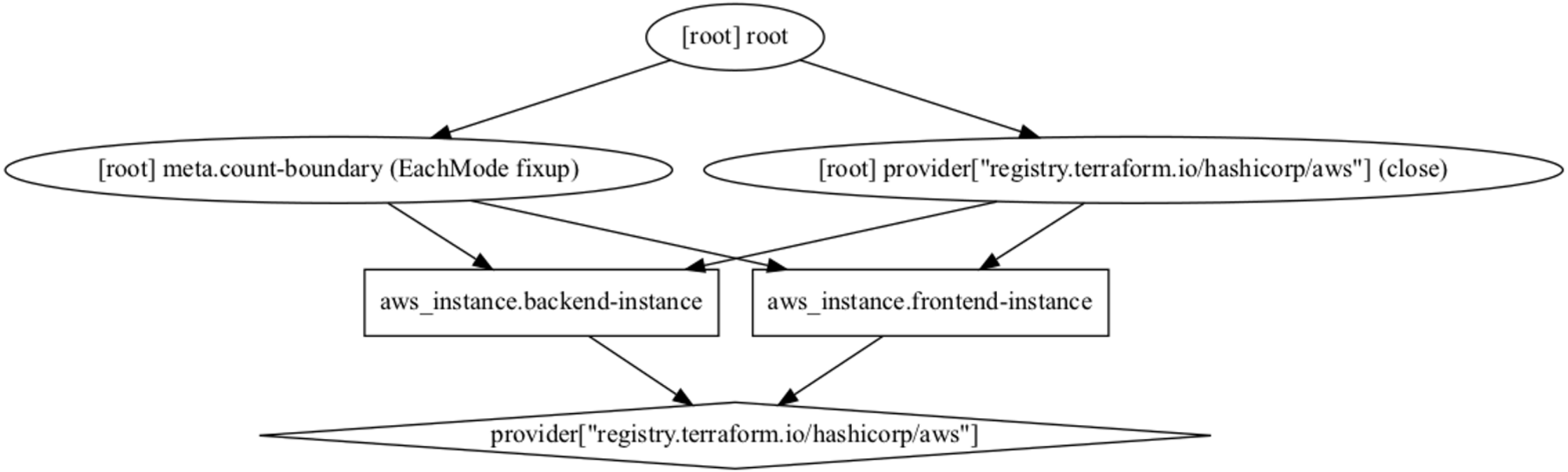
Terraform Graph Visualization

- Dot-Files are Text-Based
- Use Graphviz to visualize
- Possible output
  - Several pixel images
  - SVG
  - PDF
  - Postscript



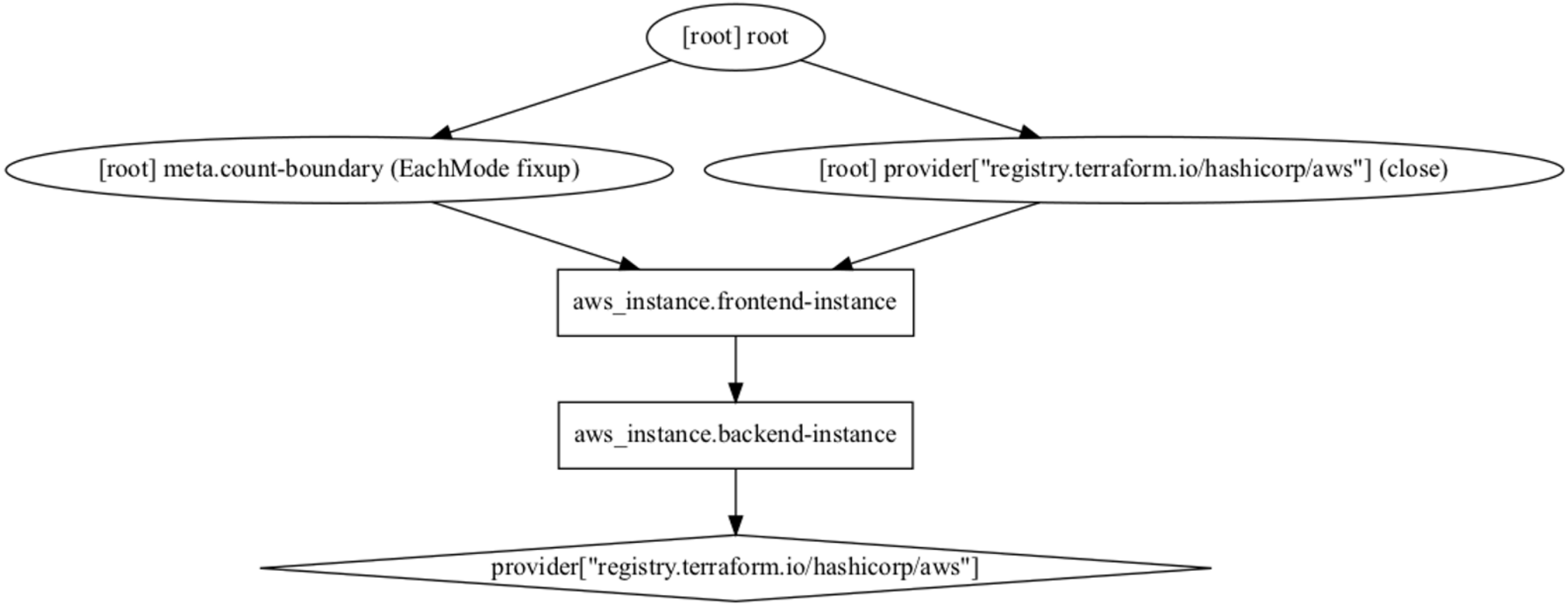
No dependency graph representation

```
$ terraform graph | dot -Tpng > no-dep-graph.png
```



Dependency graph representation

```
$ terraform graph | dot -Tpng > dep-graph.png
```



LAB

dependencie visualization

- create a second ec2 ressource in terraform
- create a graph `no-dep.png`
- change the second ec2 ressource with `depends_on`
- create a graph `dep.png`

MOD 08 - S3 Storage and Terraform

S3 Bucket-Ressource

```
resource "aws_s3_bucket" "bucket" {
  bucket = var.s3bucket_name
  acl    = "private" # Default-Value

  versioning {
    enabled = true
  }
}
```

S3 Bucket Upload

```
resource "aws_s3_bucket_object" "video_upload_object" {
  bucket = var.s3bucket_name
  key    = "video.mp4"
  source = "video.mp4"

  # The filemd5() function is available in Terraform 0.11.12 and later
  # For Terraform 0.11.11 and earlier, use the md5() function and the file() function:
  # etag = "${md5(file("path/to/file"))}"
  etag = filemd5("video.mp4")

  depends_on = [
    aws_s3_bucket.bucket
  ]
}
```

LAB

S3 essentials

- create a unique-bucket
- upload a file to it

use S3 lifecycle

```
resource "aws_s3_bucket" "bucket" {
  bucket = "terraform-20181219040316452900000001"
  acl = "private"

  lifecycle_rule {
    enabled = true
    transition {
      days = 30
      storage_class = "STANDARD_IA"
    }
    transition {
      days = 60
      storage_class = "GLACIER"
    }
  }
}
```

after 30 days move the objects to `STANDARD_IA` and after 60 days to `GLACIER`.

S3 Hosting a Website

```
resource "aws_s3_bucket" "static_bucket" {
  bucket = var.static_bucket_name
  acl    = "public-read"

  website {
    index_document = "index.html"
    error_document = "error.html"
  }
}
```

Create a Policy with Placeholders

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Allow Public Access to All Objects",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::${bucket_name}/*"
    }
  ]
}
```

S3 Create a Policy / using templatefile()

```
resource "aws_s3_bucket_policy" "bucket_policy" {
  bucket = aws_s3_bucket.static_bucket.id

  policy = templatefile("bucket_policy.tpl", {
    bucket_name = var.static_bucket_name
  })
}
```

Upload a website-Assets

```
resource "aws_s3_bucket_object" "html_object" {
  bucket       = var.static_bucket_name
  key          = "index.html"
  source       = "index.html"
  content_type = "text/html"

  # The filemd5() function is available in Terraform 0.11.12 and later
  # For Terraform 0.11.11 and earlier, use the md5() function and the file() function:
  # etag = "${md5(file("path/to/file"))}"
  etag = filemd5("index.html")

  depends_on = [
    aws_s3_bucket_policy.bucket_policy,
    aws_s3_bucket.static_bucket
  ]

}
```

LAB

Hosting a website on S3

- create an unique-bucket
- create a bucket-policy
- upload an index.html
- try to access the html-page

Use a random-Provider for unique Buckets

define the random-provider (provider.tf)

```
provider "random" {
}
```

create a random-ressource (rand.tf)

```
resource "random_uuid" "s3_random_id" {
}
```

use the random-ressource with a bucket (s3.tf)

```
resource "aws_s3_bucket" "bucket" {
  bucket = "${var.s3_bucket_name}-${random_id.s3_random_id.result}"
}
```

LAB

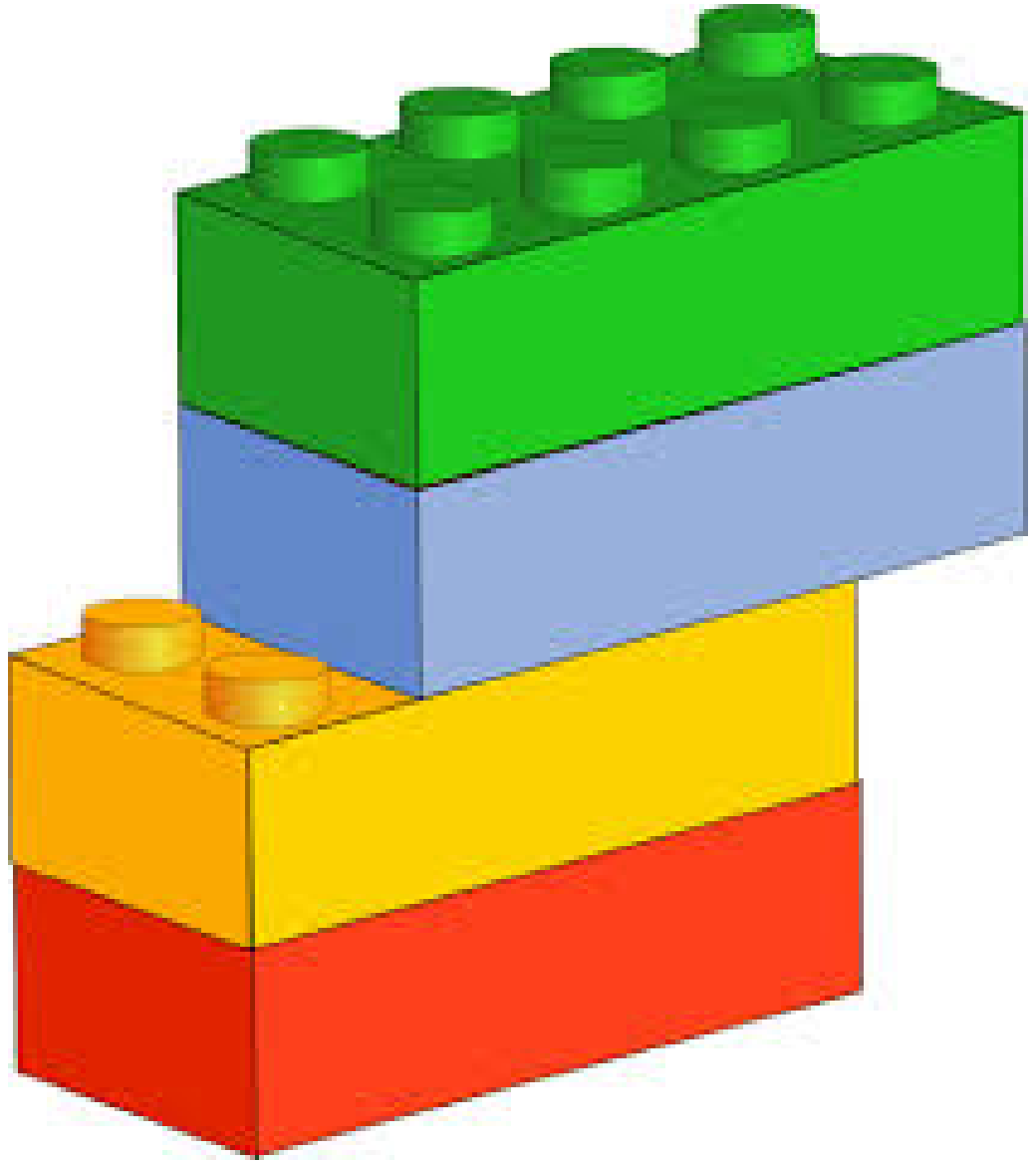
refactor with random

- use the random-provider
- implement a random-bucket-name

MOD 09 - Terraform and Modules

the power of modules

A module is a container for multiple resources that are used together. Modules can be used to create lightweight abstractions, so that you can describe your infrastructure in terms of its architecture.



vpc-module example

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.7.0"

  name = "my-vpc"
  cidr = "10.0.0.0/16"
  azs             = ["${var.region}a", "${var.region}b", "${var.region}c"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  public_subnets  = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]
  enable_vpn_gateway = false
}
```

and don´t forget to `$ terraform init`

LAB

Setup your VPC (Module-Support)

- create a VPC in `eu-central-1`
- create three public & privat Subnets with each AZ
- create an Internet-Gatway
- create a NAT-Gateway
- create a Route Table

MOD 10 - Terraform and Elastic Load Balancing

AWS Elastic Load Balancing

- automatically distributes incoming traffic across multiple targets
  - EC2 instances
  - containers
  - IP addresses
- targets one or more Availability Zones
- monitors the health of its targets
- routes traffic only to healthy targets
- scales load balancer if incoming traffic changes over time

Application Load Balancer

- internet-/internal-facing
- Layer 4
- fully managed by AWS
- routable to private Subnets
- SSL-Offload possible
- HTTP/2 Proxy possible
- uses Target-Groups and Healthchecks

Classic-Loadbalancer Ressource (simple)

```
resource "aws_elb" "elb-appserver" {
  name               = "elb-appserver"
  availability_zones = ["us-west-2a", "us-west-2b", "us-west-2c"]
  instances          = aws_instance.app_server.*.id
  security_groups    = [aws_security_group.elb-appserver-sg.id]

  listener {
    instance_port     = 80
    instance_protocol = "http"
    lb_port           = 80
    lb_protocol       = "http"
  }

  tags = {
    Name = "terraform-elb"
  }
}
```

Application-Loadbalancer Ressource

```
resource "aws_alb" "alb" {
  name            = "terraform-example-alb"
  security_groups = [aws_security_group.alb.id]
  subnets         = ["subnet-04c352b2400a7c891",
                     "subnet-0d752b61d8c0072d6",
                     "subnet-0b516a2ea3f89bdf4"]
}
```

```
resource "aws_alb_listener" "listener_http" {
  load_balancer_arn = aws_alb.alb.arn
  port              = "80"
  protocol          = "HTTP"
  default_action {
    target_group_arn = aws_alb_target_group.web-group.arn
    type             = "forward"
  }
}
```

Target-Group

Each target group is used to route requests to one or more registered targets. When you create each listener rule, you specify a target group and conditions. When a rule condition is met, traffic is forwarded to the corresponding target group.

```
resource "aws_alb_target_group" "web-group" {
  name     = "terraform-example-alb-target"
  port     = 80
  protocol = "HTTP"
  vpc_id   = var.vpc_id
  health_check {
    path = "/"
    port = 80
  }
}
```

Attach Instances to a Target-Group

This ressource does the work of adding EC2 Intances to the Target-Group of the LoadBalancer.

```
resource "aws_lb_target_group_attachment" "alb-attachment" {
  count            = var.node_count
  target_group_arn = aws_alb_target_group.web-group.arn
  target_id        = aws_instance.webserver-instance[count.index].id
  port             = 80
}
```

LAB

create an alb + webserver cluster

- create 2 webserver replicas with metadata-instance-id
- add ingress-rule for port 80 open to the world
- create an internet-facing alb with target-groups
- add a DNS-Entry for the ALB (optional)

MOD 11 - Terraform and Scaling

AWS Auto Scaling Service

Auto Scaling groups are collections of Amazon EC2 instances that enable automatic scaling and fleet management features. These features help you maintain the health and availability of your applications.

Launch configuration

Defines an instance configuration template. The Auto Scaling group uses it to launch EC2 instances.

```
resource "aws_launch_configuration" "launch_config" {
  name_prefix     = "terraform-example-web-instance"
  image_id        = "ami-04c921614424b07cd"
  instance_type   = "t2.micro"
  key_name        = aws_key_pair.my-pub-key.key_name
  security_groups = [aws_security_group.allow_web_from_anywhere.id]
  user_data       = file("install_webserver.sh")

  lifecycle {
    create_before_destroy = true
  }
}
```

Auto Scaling Goups (ASG)

```
resource "aws_autoscaling_group" "autoscaling_group" {
  launch_configuration = aws_launch_configuration.launch_config.id
  min_size             = var.autoscaling_group_min_size
  max_size             = var.autoscaling_group_max_size
  target_group_arns    = [aws_alb_target_group.web-group.arn]
  vpc_zone_identifier  = [module.vpc.private_subnets.0,
                          module.vpc.private_subnets.1,
                          module.vpc.private_subnets.2]

  tag {
    key                 = "Name"
    value               = "terraform-example-autoscaling-group"
    propagate_at_launch = true
  }
}
```

LAB

auto-scale web-server

- extend the example with ASG
- min 2 / max 5 Instances

MOD 12 - Terraform and RDS

AWS RDS Databases

- managed PaaS by AWS
- scale up on demand
- Multi-AZ / Read-Replica
- Backups managed by AWS
- EC2 underneath (one exception)

MySQL RDS Example

```
resource "aws_db_instance" "rds_mysql" {
  allocated_storage      = 20
  max_allocated_storage  = 100
  storage_type           = "gp2"
  engine                 = "mysql"
  engine_version         = "5.7"
  instance_class         = var.db_instance_type
  name                   = var.db_name
  username               = var.db_user
  password               = var.db_password
  parameter_group_name   = "default.mysql5.7"
  skip_final_snapshot    = true
  vpc_security_group_ids = [aws_security_group.mysql-access.id]
  publicly_accessible    = true
}
```

LAB

Setup a MySQL RDS

- create a mysql database
- create a security-group
- create acess to public
- test the access via db-client (optional)

AWS Aurora

- MySQL and PostgreSQL-compatible database
- five times faster than MySQL
- three times faster than PostgreSQL
- built native on AWS
- provisioned **and** serverless
- **NO** public-access when serverless!

AWS Aurora Serverless via Module 1/3

As an alternative for the `aws_rds_cluster` -Ressource, we use a Module for the serverless approach.

```
module "aurora_postgresql" {
  source  = "terraform-aws-modules/rds-aurora/aws"
  version = "6.1.4"
  name                        = "serverless-aurora-postgresql"
  engine                      = "aurora-postgresql"
  engine_mode                 = "serverless"
  storage_encrypted           = true
  master_username             = "admin"
  master_password             = "Admin123"
  create_random_password = false
  vpc_id                    = module.vpc.vpc_id
  subnets                   = module.vpc.database_subnets
  create_security_group = true
  ...
```

AWS Aurora Serverless via Module 2/3

```
...
  allowed_cidr_blocks = module.vpc.private_subnets_cidr_blocks
  monitoring_interval = 60
  enable_http_endpoint= true # cool for querying via HTTP
  apply_immediately   = true
  skip_final_snapshot = true

  db_parameter_group_name         = aws_db_parameter_group.example_postgresql.id
  db_cluster_parameter_group_name = aws_rds_cluster_parameter_group.example_postgresql.id
  # enabled_cloudwatch_logs_exports = # NOT SUPPORTED

  scaling_configuration = {
    auto_pause               = true # cluster can be paused when idle
    min_capacity             = 2    # capacity units
    max_capacity             = 8    # capacity units
    seconds_until_auto_pause = 300  # time before cluster is paused
    timeout_action           = "ForceApplyCapacityChange"
  }
}
```

AWS Aurora Serverless via Module 3/3

```
resource "aws_db_parameter_group" "example_postgresql" {
  name        = "${var.demo-name}-aurora-db-postgres-parameter-group"
  family      = "aurora-postgresql10"
  description = "${var.demo-name}-aurora-db-postgres-parameter-group"
}

resource "aws_rds_cluster_parameter_group" "example_postgresql" {
  name        = "${var.demo-name}-aurora-postgres-cluster-parameter-group"
  family      = "aurora-postgresql10"
  description = "${var.demo-name}-aurora-postgres-cluster-parameter-group"
}
```

MOD 13 - Terraform and Serverless

Lambda-Functions

- run code without infrastructure
- automatically scale on demand
- pay-as-you-use-Modell
  - by seconds of cpu
  - by amount of memory
- upload code as
  - a .zip file
  - container image

Lambda essential Ressource

```
resource "aws_lambda_function" "http_crud_lambda" {
  filename         = "lambda_function.zip"
  function_name    = "http-crud-tutorial-function"
  role             = aws_iam_role.iam_for_lambda_tf.arn
  handler          = "index.handler"
  source_code_hash = data.archive_file.lambda_zip.output_base64sha256
  runtime          = "nodejs14.x"
  architectures    = ["arm64"]
}
```

DynamoDB

- fully managed by aws
- key-value and document database
- delivers predictable performance at scale

DynamoDB Ressource

```
resource "aws_dynamodb_table" "http-crud-tutorial-table" {
  name           = "http-crud-tutorial-items"
  hash_key       = "id"
  billing_mode   = "PAY_PER_REQUEST"

  attribute {
    name = "id"
    type = "S"
  }
}
```

AWS API-Gateway

- fully managed service
- "front door" for applications
- RESTful APIs
- WebSocket APIs
- supported workloads
    - containerized
    - serverless
    - web applications

API-Gateway essential Ressources

```
resource "aws_apigatewayv2_api" "http-crud-tutorial-api" {
  name          = "http-crud-tutorial-api"
  protocol_type = "HTTP"
}
```

```
resource "aws_apigatewayv2_route" "route-1" {
  api_id    = aws_apigatewayv2_api.http-crud-tutorial-api.id
  route_key = "GET /items/{id}"
  target    = "integrations/${aws_apigatewayv2_integration.http-crud-tutorial-api.id}"
}
```

LAB

Setup a API

- create a dynamo-db table
- create a lambda function
- create an api-gateway
- test the api with `curl`

MOD 14 - EKS with Terraform

AWS Elastic Kubernetes Service

- a managed service to run Kubernetes
- runs and scales Kubernetes across multiple AZs
- can be node-based or serverless (fargate)

EKS Ressources

```
TODO: CODE HERE
```

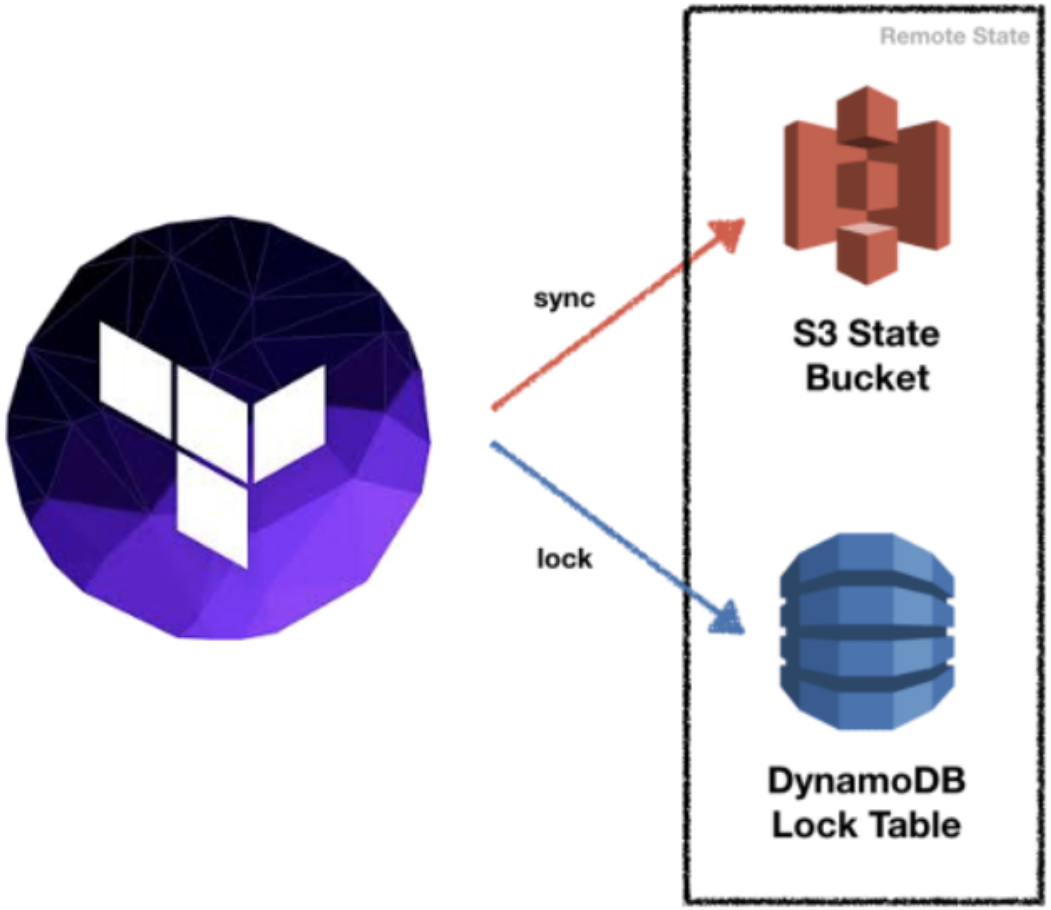
How to `kubectl` into the EKS-cluster

```
# get the kubeconfig-Data
$ aws eks --region $(terraform output -raw region) update-kubeconfig --name $(terraform output -raw cluster_name)

# check the nodes in the cluster
$ kubectl cluster-info

# check the nodes in the cluster
$ kubectl get nodes

# check if pods are running
$ kubectl get pods
```

MOD 15 - Remote State / Workspaces

Remote State File

- Stores the state at a key in a bucket on S3
- Lock the State via Dynamo DB



Backend Configuration with S3 & DynamoDB

```
terraform {
  backend "s3" {
    bucket         = "s3-terraform-backend"
    key            = "example-app/terraform.tfstate"
    region         = "eu-central-1"
    encrypt        = true
    dynamodb_table = "terraform-state-lock-dynamo"
  }
}
```

To start the initialization/migration use:

```
$ terraform init
```

bootstrap the backend-resources (optional)

```
resource "aws_s3_bucket" "s3-state" {
  bucket = "s3-terraform-backend"
  acl = "private"
}
```

```
resource "aws_dynamodb_table" "dynamodb-terraform-state-lock" {
  name = "terraform-state-lock-dynamo"
  hash_key = "LockID"
  read_capacity = 20
  write_capacity = 20

  attribute {
    name = "LockID"
    type = "S"
  }
}
```

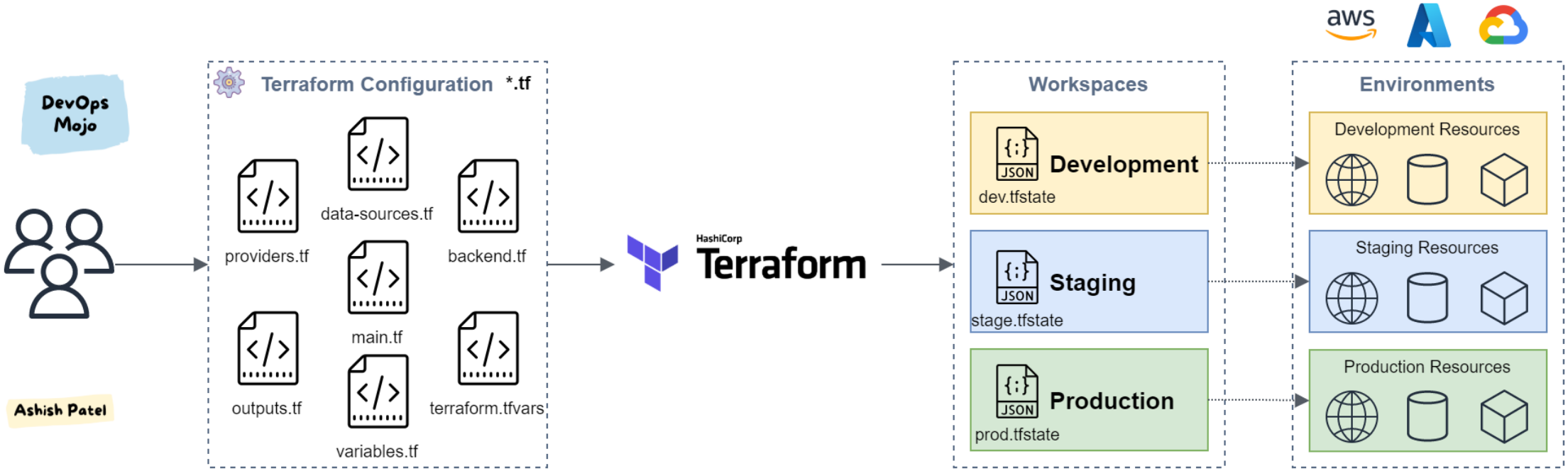
LAB

Setup remote state

- create a s3-bucket
- create a dynamodb-table with LockID
- configure the s3-backend in terrafrom
- Test it :)

Workspaces

With a remote backend and locking, collaboration is no longer a problem.



Source

Isolating State Files

However, there is still one more problem remaining: **isolation**.

Isolate State via workspaces

```
$ terraform workspace show
$ terraform workspace new production
$ terrafrom apply -var-file=env/production.tf
$ terraform workspace new development
$ terrafrom apply -var-file=env/development.tf
$ terraform workspace list
$ terraform workspace select production
```

LAB

manage workspaces

- create a remote backend (opt.)
- configure two different workspaces
  - production
  - development

MOD 15 - Terraform and Route53

The Route53 Record Resource

```
resource "aws_route53_record" "tf-alb-record" {
  zone_id = data.aws_route53_zone.zone.zone_id
  name    = "tf-alb.${var.route53_hosted_zone_name}"
  type    = "A"
  alias {
    name                   = aws_alb.alb.dns_name
    zone_id                = aws_alb.alb.zone_id
    evaluate_target_health = true
  }
}
```

How to obtain hosted zone information

you can create a new hosted zone (not usual)

```
resource "aws_route53_zone" "my-test-zone" {
  name = "example.com"
  vpc {
    vpc_id = "${var.vpc_id}"
  }
}
```

you can use a datasource here to finde pre-deployed information

```
data "aws_route53_zone" "zone" {
  name = "aws.zhtraining.de"
}
```

LAB

use Route53 (DNS)

- use hosted zone info
- create a friendly dns entry for the static s3-bucket
- try that on your browser

Thank you