Terraform 1.3 with AWS

Infrastructure as Code

Version: 1.2.0

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

Course Details 2/2

- Module 06 - From static to dynamic deployment
- Module 07 - Dependencies and Terraform
- Module 08 - Terraform an Modules
- Module 09 - Terraform and RDS
- Module 10 - Remote State

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

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)

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

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.

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"
      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"
      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)

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

Terraform *Standard* Filelayout

File / Folder	Purpose
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
- 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.

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 = "ABCDEFGXXXXXXXXXXXXXX"
}
```

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
}
```

create a keypair

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}"
  }
}
```

LAB

more attributes

create an EC2-Instance

- add a subnet-id from your VPC
- deploy a basic webserver
- create two instances
- 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
- test the file with

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

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

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

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"

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"
  }
}
```

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

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 - 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 09 - 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)

MOD 10 - 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 :)

Thank you