PRODYNA

Terraform

Training

Agenda

- Basic
  - What is Terraform?
  - Providers
  - Build, Update, Destroy
  - Input & Output Variables
  - Modules
- Advanced
  - Work in a team
  - File Structure
  - CI/CD
  - Misc.

Introduction

Dennis Creutz
Managing IT Consultant / Cloud Architect

Introduction

- Name
- What you do?
- Experience with Cloud, Terraform, etc.
- Expectations

Overview

Infrastructure as Code

Infrastructure as code is the process of managing and provisioning computer data centers through machine-readable definition files

Infrastructure as Code - Advantages

- Enables automation
  - Enables CI/CD of infrastructure
- Cost reduction
- Increased speed
- Risk reduction

IaC Tools

- Chef
- Puppet
- Ansible
- CloudFormation, Azure Resource Manager, etc...
- Terraform

Terraform

- Open-source infrastructure as code software
- Created by HashiCorp
- Written in HCL or JSON
- Supports many providers (AWS, GCP, Azure, Kubernetes, etc.)

Install Terraform

- MacOS:
  brew install terraform
- Windows:
  https://www.terraform.io/downloads.html
- Verify installation:
  terraform –version
- You need Terraform >= v1.0

Basics

Terraform Provider

- Responsible for creating and managing resources
- Translator from HCL/JSON to API interactions
- Multiple providers in one Terraform file possible

Terraform Provider

- AWS
- Azure
- GCP
- Kubernetes
- Helm
- MySQL
- Grafana
- CloudFlare
- Many many more: https://www.terraform.io/docs/providers/index.html

Terraform Resource

Resource that exists within the infrastructure (e.g. EC2 instance)

```
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

„aws_vpc“ = Resource type, defined by the provider

„example“ = Resource name, defined by you and used as Terraform internal referenz

„cidr_block“ = Resource (specific) attributes

Terraform commands

- Terraform will read all *.tf in a directory
  - Best practice: Always start with a main.tf
- Command: *terraform init*
  - Initializes various local settings and data needed by other commands
  - Especially: Downloads all provider binaries
- Command: *terraform plan [-out=myplan.tfplan]*
  - Creates a plan to visualise changes to the current infrastructure
  - No changes are applied!
  - Can be used as basis for the *terraform apply* command

Terraform commands

- Command: *terraform apply*
  - Creates a plan to visualise changes to the current infrastructure
  - Applies changes to your infrastructure (after you confirmed the changes)
- Command: *terraform destroy*
  - Destroys all resources managed by Terraform
- Command: *terraform fmt*
  - Formats all *.tf files in a directory
  - Can be used recursive with the “–recursive” flag

Demo & Practice: Basics

AWS CLI v2 - Install

- macOS:
  - brew install awscli
- Windows:
  - https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2-windows.html
- Verify:
  - aws --version
  - You need version >= 2.0

Terraform with AWS

Use AWS credentials in Terraform:

- Login into AWS console
- Use your own user or create a new terraform user with programmatic access and administrator policy
- Download credentials
- Configure AWS CLI localy with *aws configure*
  - Use the credentials from your user
  - Region: eu-central-1

Terraform with AWS

- Create directory “my-tf-first-steps”
- Create a directory “shared-credentials”
- Create a file “aws-shared-credentials”:

```
[myProfileName]
aws_access_key_id = XXXX
aws_secret_access_key = XXXX
```

Terraform with AWS

- Create file “main.tf”
- Add AWS provider:

```
provider "aws" {
  shared_credentials_file = “path/to/my/shared-credentials/file"
  profile                 = “myProfileName"
  region                  = "eu-central-1"
}
```

Demo: Create

In „main.tf“:

- Add a AWS VPC:

```
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

- *terraform init*
- *terraform apply*

Terraform State

- “terraform.tfstate”
- Maps real world resources to your configuration
- Delete state = Resources still exists but not managed by Terraform anymore
- Never interact with the state file directly!

Demo: Update

- Display your Terraform state:
  terraform show
- Change the CIDR of your VPC to 10.0.0.0/20
- Create a Terraform plan:
  terraform plan –out=myExample.tfplan
- Apply plan:
  terraform apply myExample.tfplan
- Display your Terraform state:
  terraform show

Demo: Destroy

- Destroy everything
  terraform destroy
- Display Terraform State
  terraform show

Input, Output & Dependencies

Terraform Variables

Defines Variables:

```
variable "myVar" {
  default = "test"
  type = string
  description = "My example"
}
```

Type can be string, number, object, list, bool and more.
No default value = required variable

Terraform Variables

- Use Variables with „var.myVar“
  - In Strings with ${var.myVar}
- Mostly used in Modules (later)
- Define local values:

```
locals {
  stageName = "prod"
  projectName = "prodyna-aws-training“
}
```

  Reference with „local.myLocal“
- Best practice: Define all variables in extra file „variables.tf“

Terraform Variables

Assignee Variables:

- Command-line flags:
  terraform apply -var 'myVar=test123'
- From a file (like the AWS credentials)
  - Files named *.auto.tfvars or terraform.tfvars are automatically propagated
- From environment variables (only Strings):
  TF_VAR_myVar=test123

Terraform File Structure

- main.tf
  - Provider
  - Resources
- variables.tf
  - Only for variables and locals

Terraform Output

```
output "database" {
  value = aws_db_instance.this
  sensitive = true
  depends_on = [aws_db_instance.this]
}
```

- Define outputs to..
  - Expose resource values (Modules..again)
  - Print values in terminal (terraform plan & apply)
- „sensitive“
  If true, then no output in terminal
- Attention: All resource attributes and outputs are saved in plain text in the terraform state!

Terraform File Structure

- main.tf
  - Provider
  - Resources
- variables.tf
  - Only for variables and locals
- output.tf
  - Only for outputs

Terraform Remote State & Data Source

You can access other states (read only) via data source:

```
data "terraform_remote_state" "vpc" {
    backend = "local"

    config = {
        path = "${path.module}/path/to/vpc/state/terraform.tfstate"
    }
}
```

- Data sources read data only and don't create resources
- Access output of the remote state:
  `data.terraform_remote_state.vpc.outputs.myOutputName`
- Visit the provider docs to list all available data sources

Terraform Dependencies

- Implicit:
  - Resource uses a value of another resource
- Explicit
  - Resource defines dependencies over „depends_on“

Terraform - Count

„The count parameter on resources can simplify configurations and let you scale resources by simply incrementing a number“

- Use to create multiple resources:

```
resource "aws_db_instance" "this" {
  count = 3
  name = “db-instance-${count.index}”
}
```

- Or to add a condition to the creation of the resource:

```
resource "aws_vpc" "example" {
  count = local.create_vpc ? 1 : 0
  ...
}
```

The AWS VPC will only be created if the local variable “create_vpc” is true

Demo & Practice: Input, Output & Dependencies

Terraform – Variables and Output

- Extract the variables from “main.tf” into a new file “variables.tf”
- Create a file “outputs.tf” and output the VPC values:

```
output "vpc" {
  value = aws_vpc.example
}
```

- *terraform apply*
- *terraform show*

Terraform – Count

Create multiple VPC's:

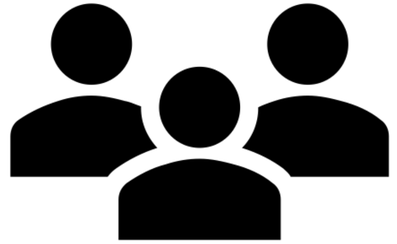
```
resource "aws_vpc" "example" {
  count = 3

  cidr_block = "10.${count.index}.0.0/20"
}
```

What will happen?

terraform apply

Practice: Basics

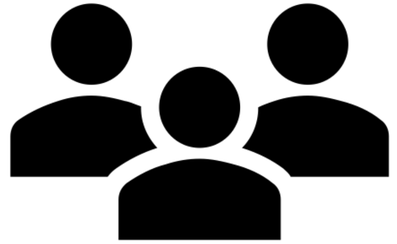
Practice 1.1: Basics



1. Create a SSH key in the AWS console (EC2-> Key Pairs)
2. Write your first Terraform code, use the AWS provider with a shared credentials file and “eu-central-1” as region
3. Create a VPC with 3 public subnets
4. Create an “t3.micro” Ubuntu EC2 instance in one of this public subnets with a Security Group that only allows SSH access and uses your SSH key

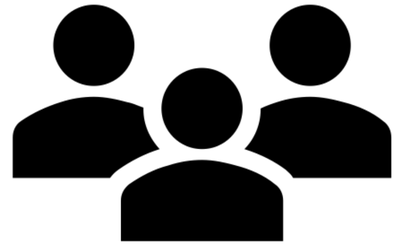
Use the Terraform AWS provider docs: https://www.terraform.io/docs/providers/aws/index.html
EC2 instance: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance

Practice 1.2: Basics - Serverless



- Create the serverless task in Terraform
- You can use “api-gateway.tf” to create your API Gateway

Serverless – Trump Quoter



Git: https://github.com/DennisCreutz/aws-training/tree/task/serverless

1. Checkout code for the task
2. Create a Lambda function "trump-quoter" with the code from the /app directory
   (don't forget to use npm install)
3. Create a REST API Gateway with a GET method that calls the Lambda function over Lambda proxy request
   => You need to enable CORS!
   => Please use regional endpoints
4. Create a S3 webhosting bucket with the HTML from the /html directory
   => Bucket name: *yourTeamName-currentDateTime*
   => Enable public access
5. Enter the API Gateway URL and a query (e.g. "Obama") in the input fields and hit "TRUMP QUOTE"

Modules

Terraform Modules

“A module is a container for multiple resources that are used together.”

- We already used a module, the root module
- Modules can use other modules:

```
module "vm" {
  source = "./ec2-vm-module"
  instances = 2
}
```

“vm” = The name of this module instance used to reference the module in Terraform

“source” = Path to the module source code (local or Git repository)

“instances” = Module specific input parameters

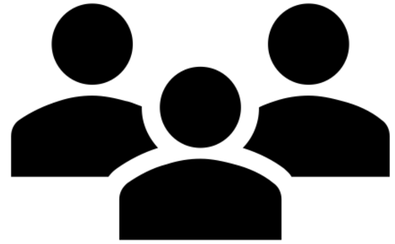
Terraform Modules

- „source“
  - Required
  - Local or remote (like Git) location
- Other parameters are defined by the modules input variables
- You can access all modules output variables: module.*moduleName*.*varName*
- ~~No built in „depends_on“ for Modules (yet)!~~
  - Since Terraform 0.13 you can use „depends_on“ with modules!

Demo: Modules

Practice: Modules

Practice 2: Modules



Extract a Lambda module from your serverless solution

1. The module should have the following variables:
    a) Tags
        1. Stage
        2. Project
    b) The Lambda timeout
    c) The Lambda function name
    d) The Lambda memory size | default to 128
    e) The Lambda runtime | default to “nodejs14.x“
    f) The Lambda handler name | default to “index.handler”
2. Which outputs do you need to define?

State manipulation

Terraform State Manipulation

- Output the Terraform state:
  terraform show
- Recreate a already created resource:
  terraform taint adressOfTheResource
- Apply/Destroy only specific resources:
  terraform apply/destroy –target=resourceName
  - Caution: Dependencies are ignored!
- Remove resource from State:
  terraform state rm adressOfTheResource
  - Resource is only removed from state but not destroyed
  - Use only as last resort!

Demo: State Manipulation

Using Terraform in a team

Using Terraform as a team

- Till now only worked on one device with a local state
- In a team we need:
  - A distributed state
  - A way to prevent multiple state manipulations at the same time
  - Decide on a file structure
  - Integrate Terraform into the build chain

Terraform Remote State

- Currently: Local state file for every team member
- Problems:
  - Possible outdated state file
  - Need to make sure no one modifies the state at the same time
- Solution: Remote state

Terraform Remote State

“With remote state, Terraform writes the state data to a remote data store, which can then be shared between all members of a team.“

Supported state stores:

- Terraform Cloud
- HashiCorp Consul
- Amazon S3
- And more...

Terraform Remote State Example

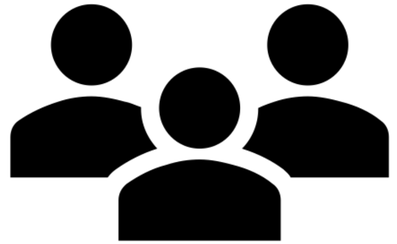
```
terraform {
    backend "s3" {
        bucket                  = “mybucket"
        region                  = "eu-central-1"
        shared_credentials_file = "../shared-credentials/aws-shared-credentials"
        profile                 = "myprofile"
        dynamodb_table          = “mydynamodb"
        key                     = "live/aws-training/prod/mycomponent/terraform.tfstate"
        encrypt                 = true
    }
}
```

Amazon S3 Remote State

- Amazon S3 for store the remote state
  - Encryption
  - Versioning
  - Access Control
  - High durability and availability
  - Serverless
- Amazon DynamoDB for state lock management

Practice: Migrate to remote state

Practice: Terraform in a Team



1. Create a remote backend store with state lock management
   - Option A:
     - Create a S3 bucket with versioning and encryption enabled
     - Create a Dynamo DB with

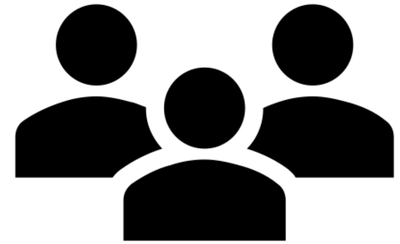
```
attribute
  name    "LockID"
  type    "S"


hash_key     = "LockID"
```

   - Option B:
     - Use this module: https://github.com/DennisCreutz/terraform-aws-remote-backend.git
       Prefix the module source with “git::” and lock the version with “?ref=myTagName”
       e.g.: git::https://github.com/DennisCreutz/terraform-aws-remote-backend.git?ref=1.0.2

Practice: Terraform in a Team

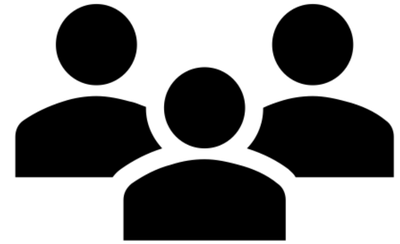
2. Migrate all local states to remote state. Example:

```
terraform {
    backend "s3" {
        bucket                  = “mybucket"
        region                  = "eu-central-1"
        shared_credentials_file = "../shared-credentials/aws-shared-credentials"
        profile                 = "myprofile"
        dynamodb_table          = “mydynamodb"
        key                     = "live/aws-training/prod/mycomponent/terraform.tfstate"
        encrypt                 = true
    }
}
```



Integrate your team name and the stage „prod“ in your remote backend keys

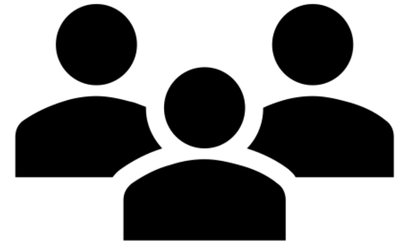
Practice: Terraform in a Team



3. Add version lock to prevent breaking changes. E.g.:

```
terraform {
  required_version = "~> 1.0"
  required_providers {
    aws = "~> 3.61"
  }
}
```

Practice: Terraform in a Team



4. Migrate your Terraform code to CodeCommit

Terraform file structure

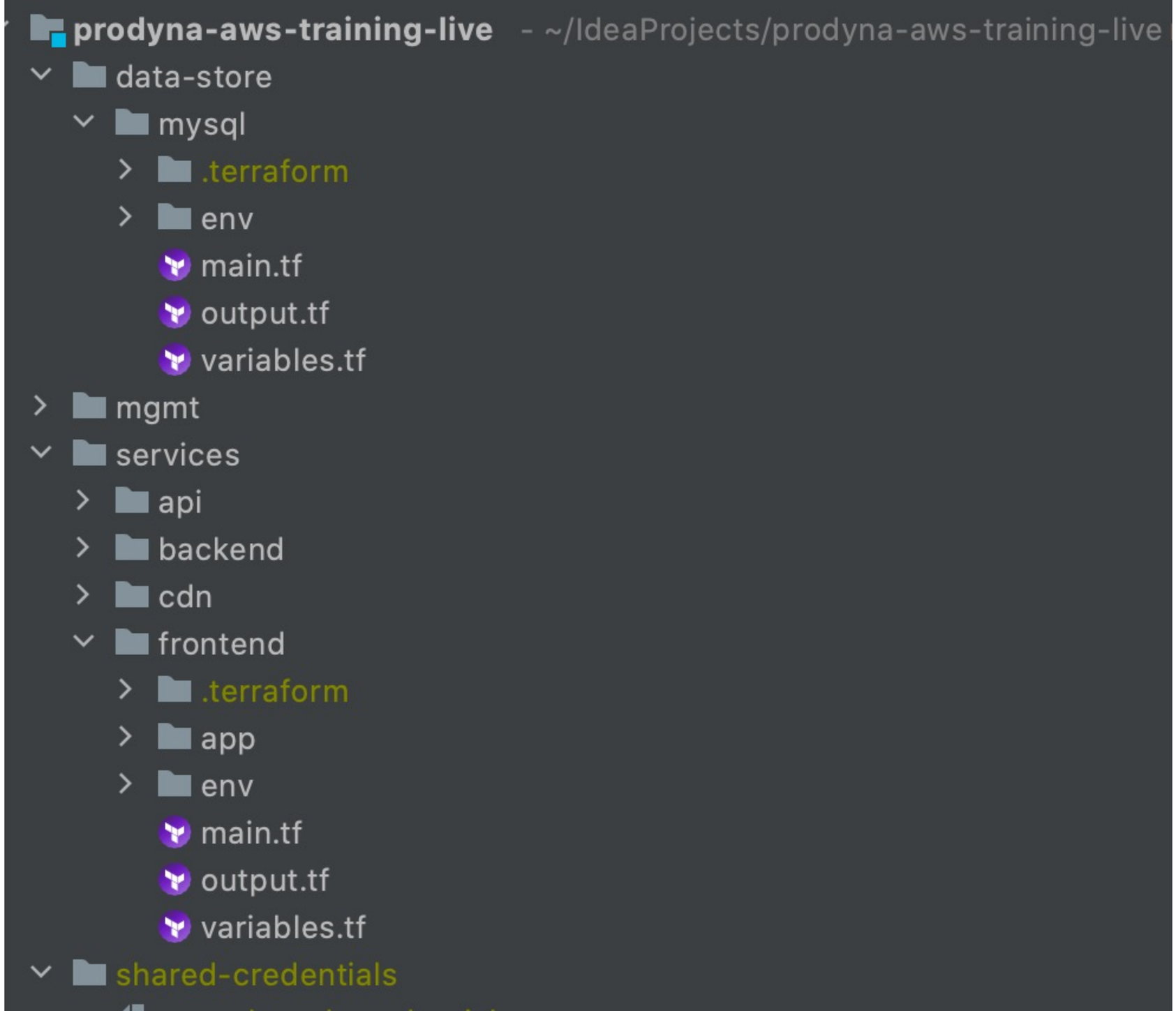
Terraform File Structure

Question: How to structure your Terraform files for multiple stages, projects and modules?

- Every project = Own git repository
- All modules in one git repository or even better: Every module got it's own repository
  - Use tags for module versioning!
  - Lock module version with "?ref=myTag":

```
module "database" {
  source = "git::https://mygitrepro.de?ref=1.1.17"
```

- Each component (data-storage, services, ...) = Own directory = Many small Terraform states
- Use modules to avoid code duplication


prodyna-aws-training-live - ~/IdeaProjects/prodyna-aws-training-live
data-store
mysql
.terraform
env
main.tf
output.tf
variables.tf
mgmt
services
api
backend
cdn
frontend
.terraform
app
env
main.tf
output.tf
variables.tf
shared-credentials

Terraform Staging

But how to integrate staging?

- Create environment directory in every component
- Create a sub-directory for every environment
- Extract remote backend configuration to a backend configuration file (e.g. backend.config)
- Extract (stage) variable assignments to variables file (e.g. terraform.tfvars)


Project
prodyna-aws-training - ~/IdeaProjects/prodyna-aws-trai
prodyna-aws-training-live - ~/IdeaProjects/prodyna-aws
data-store
mysql
.terraform
env
prod
backend.config
variables.tfvars
main.tf
output.tf
variables.tf
backend.config
1 key = "live/prodyna-aws-training/prod/data-store/mysql/terraform.t
2 shared_credentials_file = "../../shared-credentials/aws-shared-creden
3 profile = "training20"
4
variables.tfvars
1 vpc_remote_key = "live/prodyna-aws-training/prod/mgmt/v
2 stage = "prod"
3
4 # Normally it is better use a secure store (like AWS System Parameter
5 database_username = "admin1238"
6 database_password = "h68UGSmhks7asGJIj"
7 database_name = "prodynaAwsTrainingDB"
8

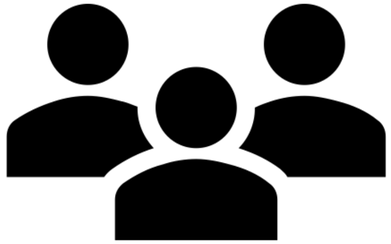
Terraform Staging

Now we can reference the correct stage in our remote backend on init:
terraform init -backend-config="./env/prod/backend.config"

And propagate the correct stage variables on plan & apply:
terraform apply -var-file="./env/prod/variables.tfvars"

Practice: Terraform file structure

Practice: File Structure



Create a “live” and “modules” directory:

1. Extract Lambda module to the new “modules” directory
2. Apply the best practice file structure to your ”live” repository
   - You can use your current code for the “prod” stage
   - Think about what you need to change to support multiple stages in your Terraform code (naming conflicts)
   - The API Gateway got a dependency to the Lambda function. Because now API Gateway and Lambda got there own state, the API Gateway need to use a “terraform_remote_state“ datasource to access the Lambda state outputs.
3. Create a “dev” stage

Remote State Datasource

```
data "terraform_remote_state" "lambda" {
  backend = "s3"

  config = {
    bucket                  = “mybucket"
    region                  = "eu-central-1"
    key                     = "my-lambda-key"
    shared_credentials_file = "../shared-credentials/aws-shared-credentials"
    profile                 = "myprofile"
  }
}
```

Terraform in your build chain

"Feature Branches" in Terraform

You want to add a new feature to your infrastructure (e.g. add a new EC2 that interacts with the database)

- Problem: Git branch from live repro. = different branch but same state file
- Solution: Terraform Workspaces
  - *terraform workspace show*
  - *terraform workspace new workspaceName*
  - *terraform workspace select workspaceName*
- Workspaces creates a new EMPTY state file
  - S3 path: env:/my-remote-backend-key
- Attention: The selected workspace is saved in the .terraform folder!
- You can now create and test your own infrastructure and destroy afterwards
- To avoid naming conflict add the workspace name to your resource names. E.g.:

```
name        = "${var.stage}-${var.project}-${terraform.workspace}-s3-frontend"
```

Integrate Terraform into your build chain

- Question: How to integrate infrastructure changes?
- Answer:
  1. Create branch
  2. Create workspace
  3. Add changes
  4. Test changes (create new resources > test > destroy resources)
  5. Select default workspace
  6. Create Terraform plan and save output:
     - terraform plan –out=myFeature.tfplan
     - Save terminal output to file (myFeature.tfplan.txt)
  7. Create PR and add *.tfplan and *.tfplan.txt
  8. Merge

Misc.

Import Resources

- You can import already created Cloud resources
- See provider docs for command
- E.g.: *terraform import aws_instance.web i-12345678*
- You need to have a resource with this name already in your Terraform file

Tools around Terraform

- Terragrunt
  - Thin wrapper for Terraform that provides extra tools for keeping your Terraform configurations DRY, working with multiple Terraform modules, and managing remote state.
- Terratest
  - Go library that makes it easier to write automated tests for your infrastructure code.
- Terraspace
  - It provides an organized structure, conventions over configurations, keeps your code DRY, and adds convenient tooling.

More under https://github.com/shuaibiyy/awesome-terraform

PRODYNA

ANY QUESTIONS?