Terraform

Agenda

PRODYNA

- Basic
  - What is Terraform?
  - Providers
  - Build, Update, Destroy
  - Input & Output Variables
  - Modules
- Advanced
  - Work in a team
  - State manipulation
  - Import Resources
  - Misc.

Introduction

Dennis Creutz
Senior IT Consultant / Software Architect

Introduction

- Name
- What you do?
- Experience with CI/CD/Terraform
- Expectations

OVERVIEW

TERRAFORM

Infrastructure as Code

PRODYNA

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

Infrastructure as Code - Advantages

PRODYNA

- 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

PRODYNA

- Open-source infrastructure as code software
- Created by HashiCorp
- Written in HCL or JSON
- Supports many providers (AWS, GCP, Azure, Kubernetes, etc.)
- Current version: 0.12

Install Terraform

- MacOS:
  brew install terraform
- Windows:
  https://www.terraform.io/downloads.html
- Verify installation:
  terraform --version

BUILD, UPDATE AND DESTROY

TERRAFORM

Terraform Provider

PRODYNA

- 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

PRODYNA

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

Terraform commands

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

Terraform commands

PRODYNA

- terraform apply
  - Creates a plan to visualise changes to the current infrastructure
  - Applies changes to your infrastructure (after you confirmed the changes)
- terraform destroy
  - Destroys all resources managed by Terraform
- terraform fmt
  - Formats all *.tf files
- terraform validate
  - Checks if the configuration is syntactically valid

DEMO & PRACTICE: CREATE

TERRAFORM

AWS CLI - Install

- macOS:
  - brew install awscli
- Windows:
  - https://docs.aws.amazon.com/de_de/cli/latest/userguide/install-windows.html
- Verify:
  - aws --version

Use AWS credentials in Terraform:

1. Login into AWS console
2. Create a user “terraform” with programmatic access and administrator policy
3. Download credentials
4. Configure AWS CLI with „aws configure“
   - Use your credentials
   - Region: eu-central-1
   - Output format: json

5. Create directory “my-tf-first-steps”
6. Create a file “aws_credentials.auto.tfvars”:

```
aws_credentials = {
  access_key = “XXXXX"
  secret_key = “YYYYYY"
}
```

7. Create file “main.tf”
8. Add credential variable (more to variables later):

```
variable "aws_credentials" {
  type = object({
    access_key = string
    secret_key = string
  })
  description = "AWS credentials used for terraform."
}
```

9. Add AWS provider:

```
provider "aws" {
  access_key = var.aws_credentials.access_key
  secret_key = var.aws_credentials.secret_key
  region = "eu-central-1"
}
```

Demo: Create

PRODYNA

In „main.tf“:

1. Add a VPC:

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

2. terraform init
3. 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 & PRACTICE: UPDATE

TERRAFORM

Demo: Update

PRODYNA

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

DEMO & PRACTICE: DESTROY

TERRAFORM

Demo: Destroy

PRODYNA

1. Destroy everything
   terraform destroy
2. Display Terraform State
   terraform show

INPUT, OUTPUT & DEPENDENCIES

TERRAFORM

Terraform Variables

PRODYNA

Defines Variables:

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

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:

1. Command-line flags:
   terraform apply -var ‘myVar=test123‘
2. From a file (like the AWS credentials)
   - Files named *.auto.tfvars or terraform.tfvars are automatically propagated
3. 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

PRODYNA

```
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 are only to read data and not to 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

PRODYNA

- 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
  ...
}
```

DEMO & PRACTICE: INPUT, OUTPUT & DEPENDENCIES

TERRAFORM

Terraform – Variables and Output

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

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

3. terraform apply
4. terraform show

PRODYNA

5. Create multiple VPC's:

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

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

6. What will happen?
7. terraform apply

PRACTICE: BASICS I

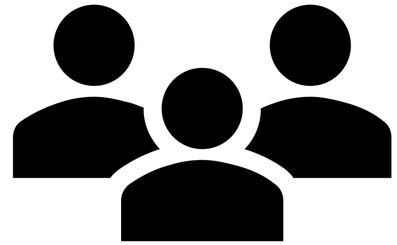
TERRAFORM

Practice 1: Convert the shopping system to Terraform

PRODYNA

Git repro: https://github.com/DennisCreutz/prodyna-aws-training

1. Use the code under “prodyna-aws-training/practices/terraform-basic-practice/” as basis
2. Copy the unfinished files and complete them.
3. Check for TODO's and missing resources.
   The recommended order is:
   1. VPC
   2. Database
   3. Backend
   4. API
   5. Frontend
4. Use the Terraform AWS provider docs: https://www.terraform.io/docs/providers/aws/index.html



Time: 120 min

MODULES

TERRAFORM

Terraform Modules

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

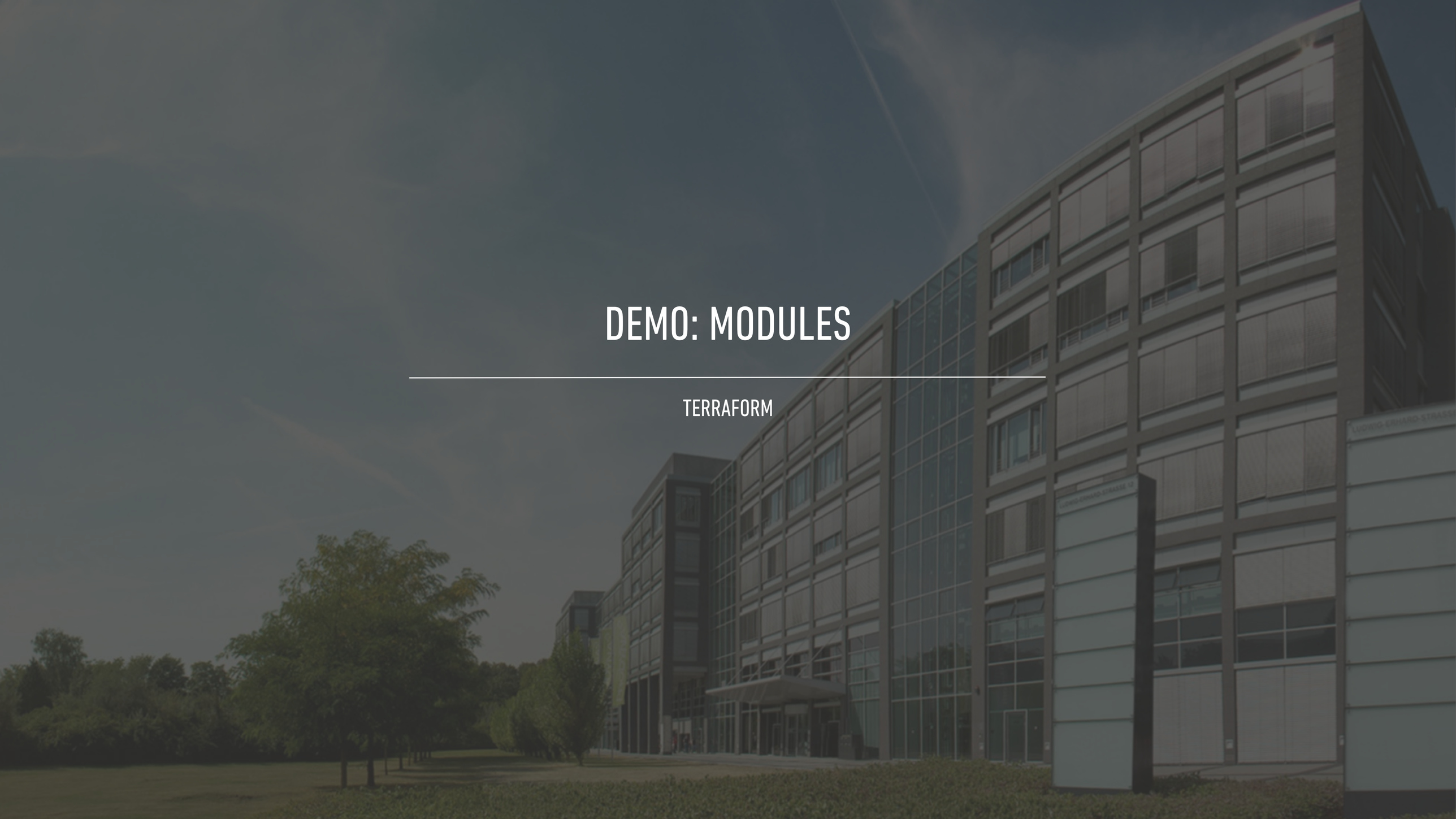
- We already used a module, the root module
- Modules can use other modules:

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

Terraform Modules

PRODYNA

- „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)!



DEMO: MODULES

TERRAFORM

PRACTICE: BASICS II

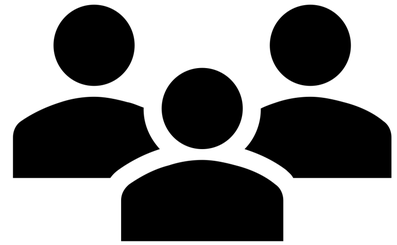
TERRAFORM

Practice 2: Modules

PRODYNA

Extract the VPC as module

- Define all needed (input) variables and outputs.
- Reference your module



Time: 45 min

STATE MANIPULATION

TERRAFORM

Terraform State Manipulation

PRODYNA

- 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

TERRAFORM

USING TERRAFORM IN A TEAM

TERRAFORM

Using Terraform as a team

PRODYNA

- 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

PRODYNA

"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...

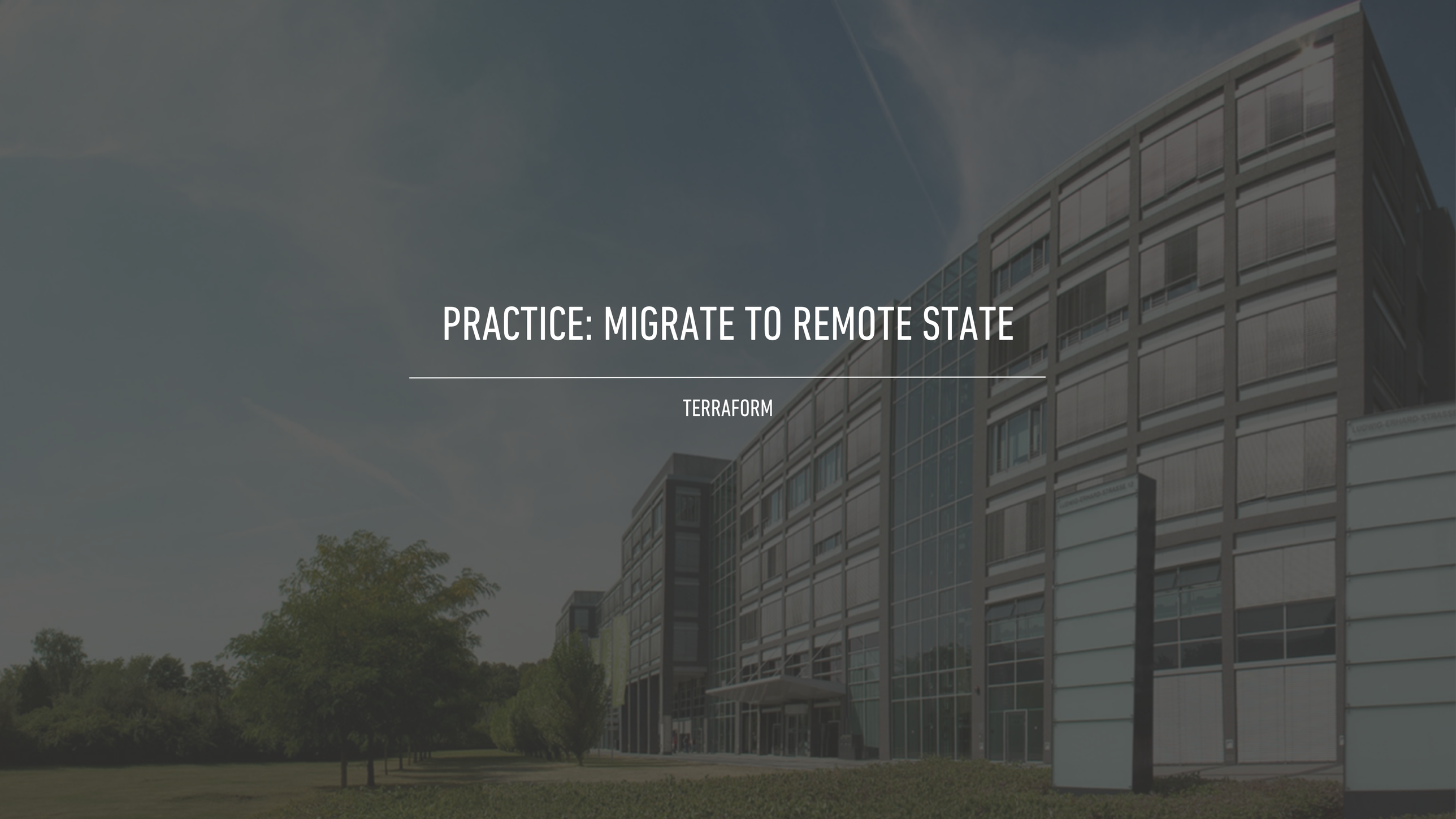
Terraform Remote State

PRODYNA

```
terraform {
  backend "s3" {
    bucket         = “myBucketName"
    key            = "live/prodyna-aws-training/prod/mgmt/dns/terraform.tfstate"
    region         = "eu-central-1"
    dynamodb_table = “myDynamoDBTable"
    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
- Know issue! Need to export AWS credentials:
  export AWS_ACCESS_KEY_ID=XXXXXXX
  export AWS_SECRET_ACCESS_KEY=YYYYYY



PRACTICE: MIGRATE TO REMOTE STATE

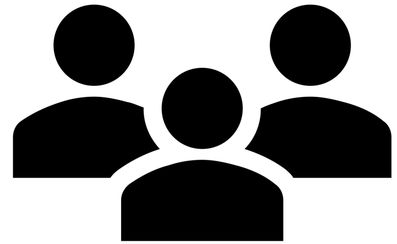
TERRAFORM

Practice: Terraform in a Team

PRODYNA

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

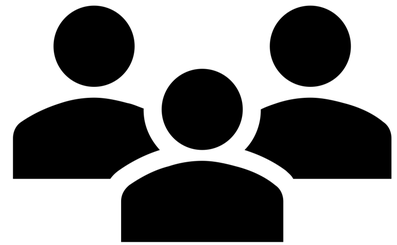
```
attribute {
  name = "LockID"
  type = "S"
}
hash_key     = "LockID“
```



Time: 45 min

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

```
terraform {
  backend "s3" {
    bucket         = “myBucketName"
    key            = "live/prodyna-aws-training/prod/mgmt/dns/terraform.tfstate"
    region         = "eu-central-1"
    dynamodb_table = “myDynamoDBTable"
    encrypt        = true
  }
}
```



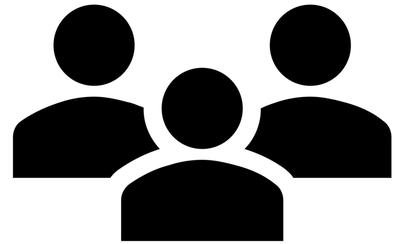
Time: 45 min

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

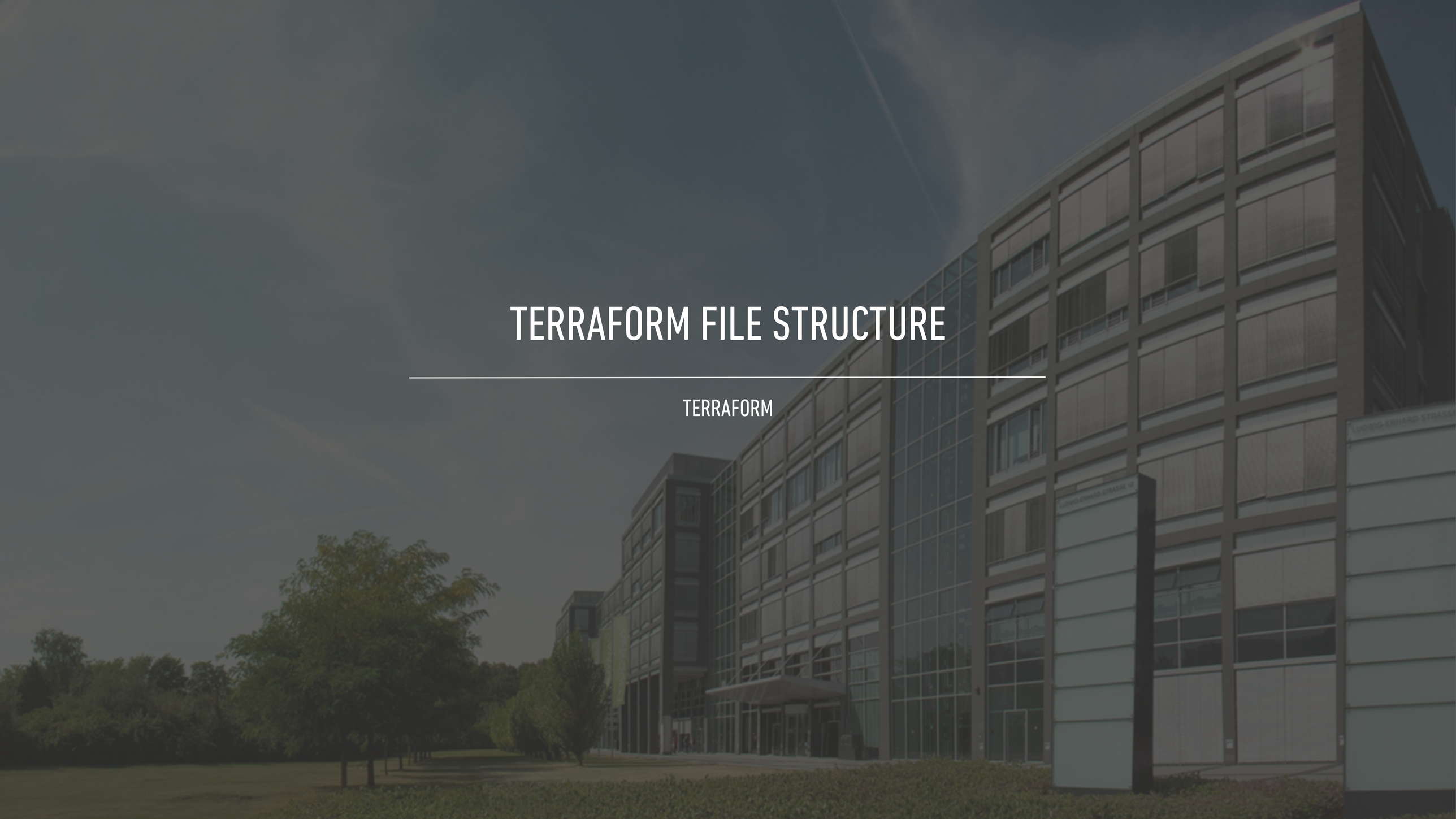
PRODYNA

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

```
terraform {
  required_version = "~> 0.12"
  required_providers {
    aws = "~> 2.47"
  }
}
```



Time: 45 min



TERRAFORM FILE STRUCTURE

TERRAFORM

Terraform File Structure

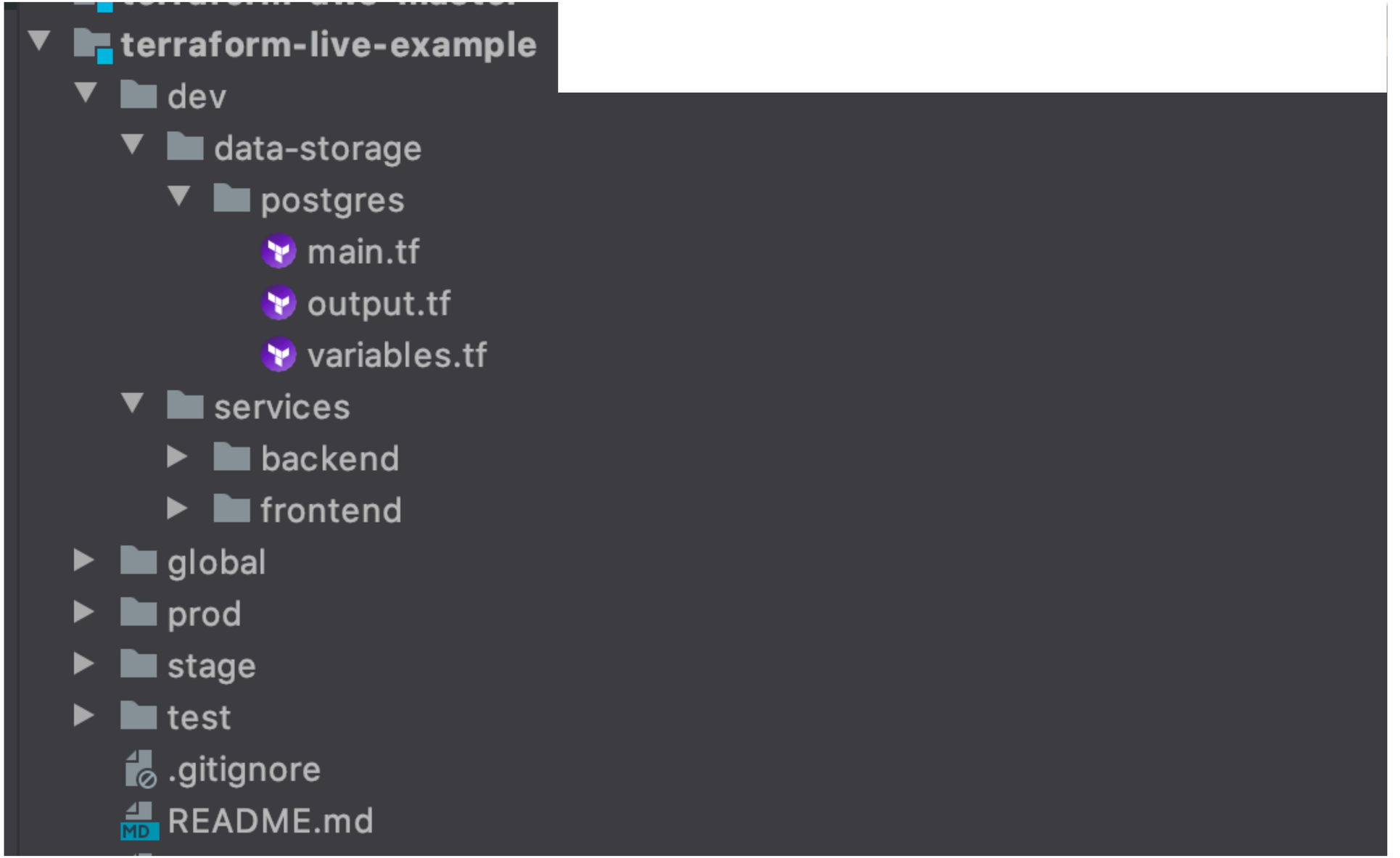
- Question: How to structure your Terraform files for multiple stages, projects and modules?
- Many answers, today only one

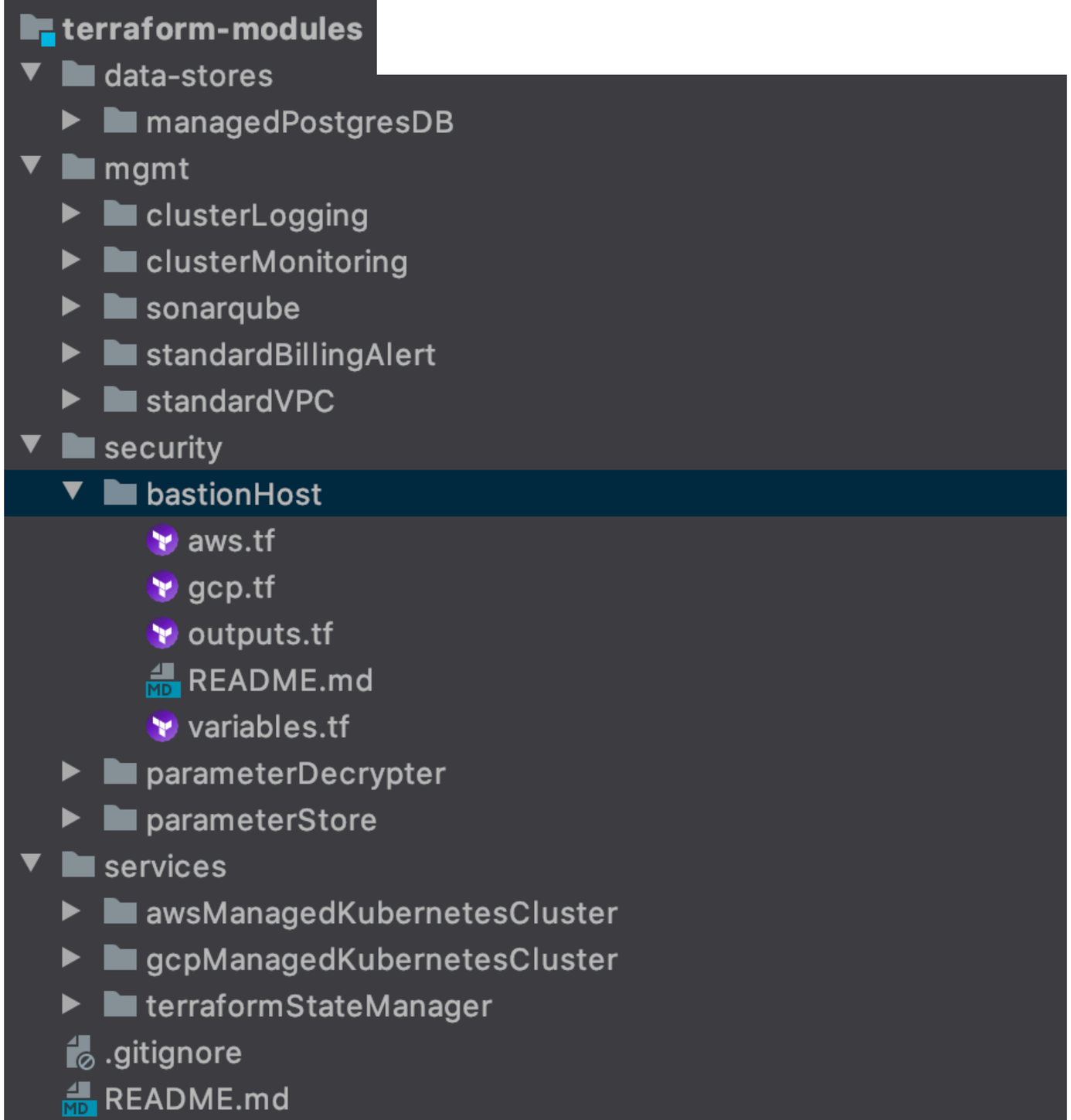
Terraform File Structure

- Every project = Own git repository
- All modules in one git repository
  - Use tags for module versioning!
  - Lock module version with “?ref=myTag”:

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

- Every stage (dev, test, prod, ...) = Own directory in repository (live repros)
- Each component (data-storage, services, ...) = Own directory in stage
- Use modules to avoid code duplication


terraform-live-example
dev
data-storage
postgres
main.tf
output.tf
variables.tf
services
backend
frontend
global
prod
stage
test
.gitignore
README.md


terraform-modules
data-stores
managedPostgresDB
mgmt
clusterLogging
clusterMonitoring
sonarqube
standardBillingAlert
standardVPC
security
bastionHost
aws.tf
gcp.tf
outputs.tf
README.md
variables.tf
parameterDecrypter
parameterStore
services
awsManagedKubernetesCluster
gcpManagedKubernetesCluster
terraformStateManager
.gitignore
README.md

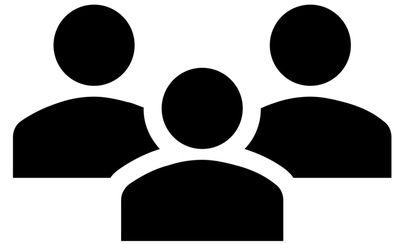
PRACTICE: TERRAFORM FILE STRUCTURE

TERRAFORM

Practice: File Structure

Create a “live” and “modules” repository:

- Extract your modules to the new “modules” repository
- Apply the best practice file structure to your ”live” repository



Time: 25 min

TERRAFORM IN YOUR BUILD CHAIN

TERRAFORM

Integrate Terraform into your build chain

PRODYNA

- 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/xxx
- 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        = "${local.stageName}-${local.projectName}-${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
  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

TERRAFORM

Import Resources

PRODYNA

- 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

Provisioners

PRODYNA

- Execute action on local or remote machine
- Should be used as last resort – Use providers if possible
- Built-in provisioners:
  - Local-exec
  - Remote-exec
  - File
  - Puppet
  - ...

Provisioners

PRODYNA

```
resource "aws_instance" "web" {

  ...
  provisioner "local-exec" {
    command = "echo The server's IP address is ${self.private_ip}"
  }
}
```

- Use „self“ to access parent resource
- Use
  when = "destroy"
  to execute provisioner only on destroy

Template Provider

„The template provider exposes data sources to use templates to generate strings for other Terraform resources or outputs.“

```
data "template_file" "init" {
  template = file("${path.module}/init.tpl")
  vars = {
    consul_address = aws_instance.consul.private_ip
  }
}
```

- „file“ function loads a file
- path.module returns the full path of the current module

Thank you

ANY QUESTIONS?