Microsoft

Terraform Core workflow and Configuration

Conditions and Terms of Use

Copyright and Trademarks

Agenda

Bigger picture around IaC

Terraform

- Core Workflow
- Config Options, Expressions, Functions
- Dependency Management and Imports
- Local and Remote Modules
- State Management
  - Remote State Management using Azure Blob Storage
- Niche Topics
  - Workspaces and Provisioners
- DevOps with Terraform

High Level Concepts

Mutable vs Immutable Infrastructure

Mutable = Can be changed after creation

Immutable = Always destroyed and re-created for changes

E.g. VM

Mutable

- Deploy with Terraform
- Update in place
- Make changes manually

Immutable

- Build template with pre-baked dependencies and app
- Deploy with Terraform
- Build new template for patching
- Deploy with Terraform (scale in and out or replace one node at a time)

IaC Provisioning vs Configuration Management

IaC Provisioning = Use an IaC Tool to deploy infrastructure (e.g. Terraform)

Configuration Management = Use a provisioning tool to configure deployed infrastructure (e.g. Ansible)

Can IaC Provisioning replace VM Configuration Management? In many cases YES with an Immutable Infrastructure approach.

Introduction to Terraform

Hashicorp Terraform

~~Open-source~~ Business Source IaC tool for provisioning

Support for all major cloud providers and inhouse

Supports higher level PaaS solutions too

Doesn’t require a server-side agent

IaC is written using Hashicorp Configuration Language (HCL)

Terraform Tooling Overview

Terraform CLI – Always Free and Full Functionality

Terraform Cloud and Terraform Enterprise – Server-Side Products

- Encapsulate Terraform CLI Runs
  - UI/VCS-driven workflow
  - API-driven workflow
  - CLI-driven workflow
- Private Registry
- Drift Detection
- Etc

CLI version 1.9 and above recommended for greenfield projects.

Terraform Core Workflow – Getting Started

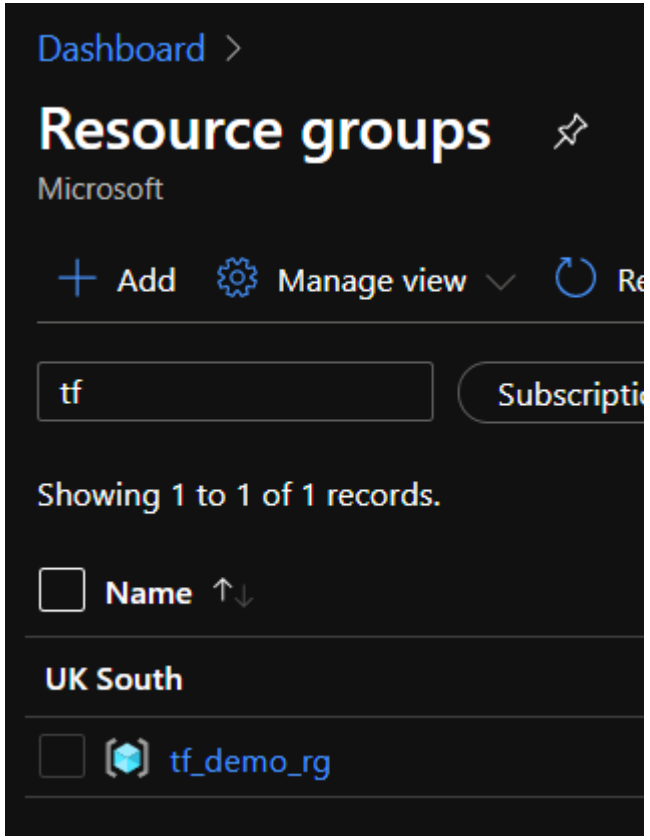
Your code becomes your infrastructure

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "= 4.7.0"
    }
  }
}

provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "example" {
  name     = "tf_demo_rg"
  location = "UK South"
}
```

terraform apply



Providers and Resources

```
# main.tf
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "= 4.7.0"
    }
  }
}

provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "example" {
  name     = "tf_demo_rg"
  location = "UK South"
}
```

type name

Providers

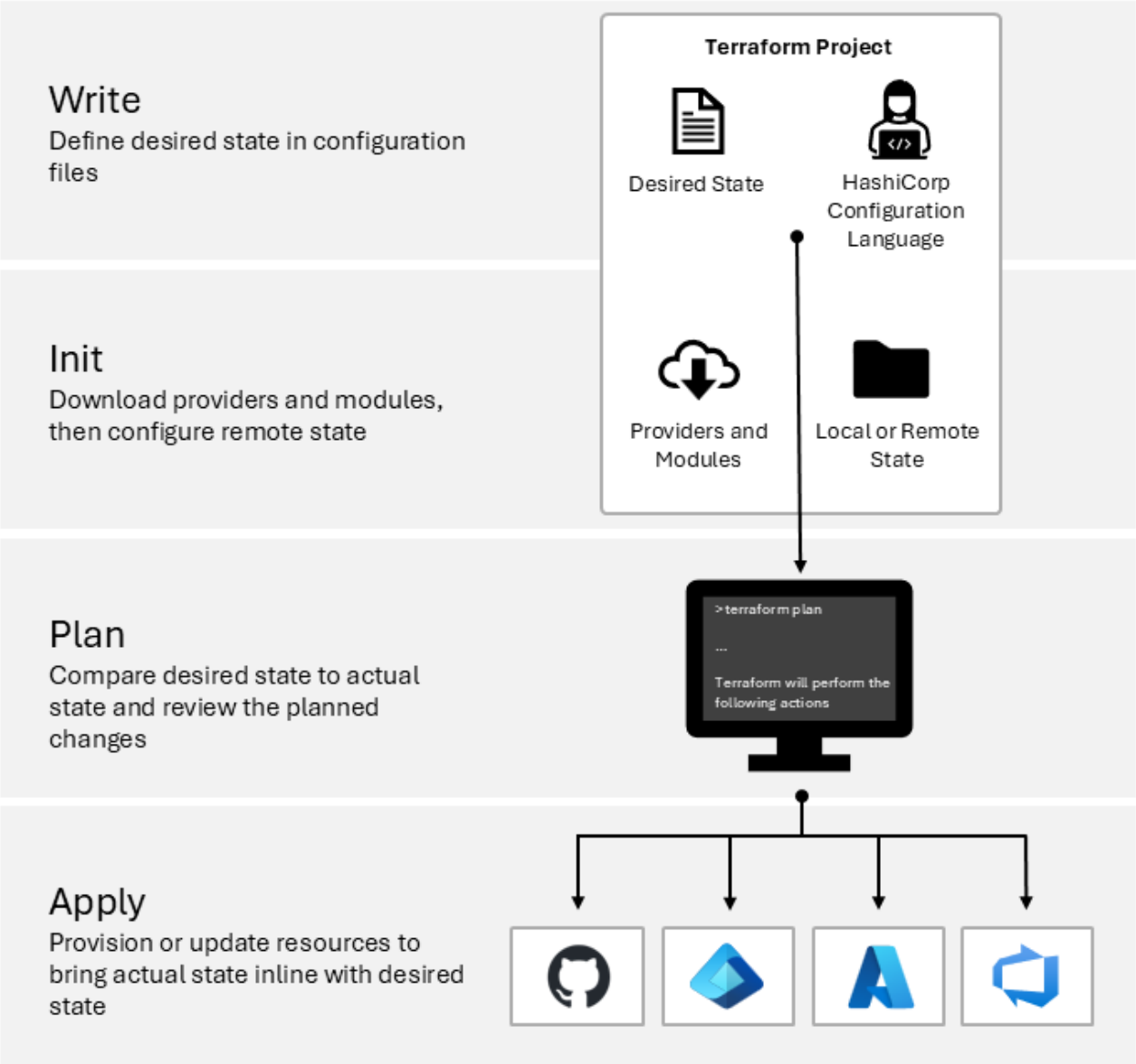
Terraform core knows nothing about target API

Providers are plug-ins. They implement auth and API CRUD operations for the target API

Microsoft owned or co-maintained providers

- hashicorp/azurerm
- azure/azapi
- hashicorp/azuread
- integrations/github
- microsoft/azuredevops
- microsoft/fabric
- Terraform Resource Modules | AVM

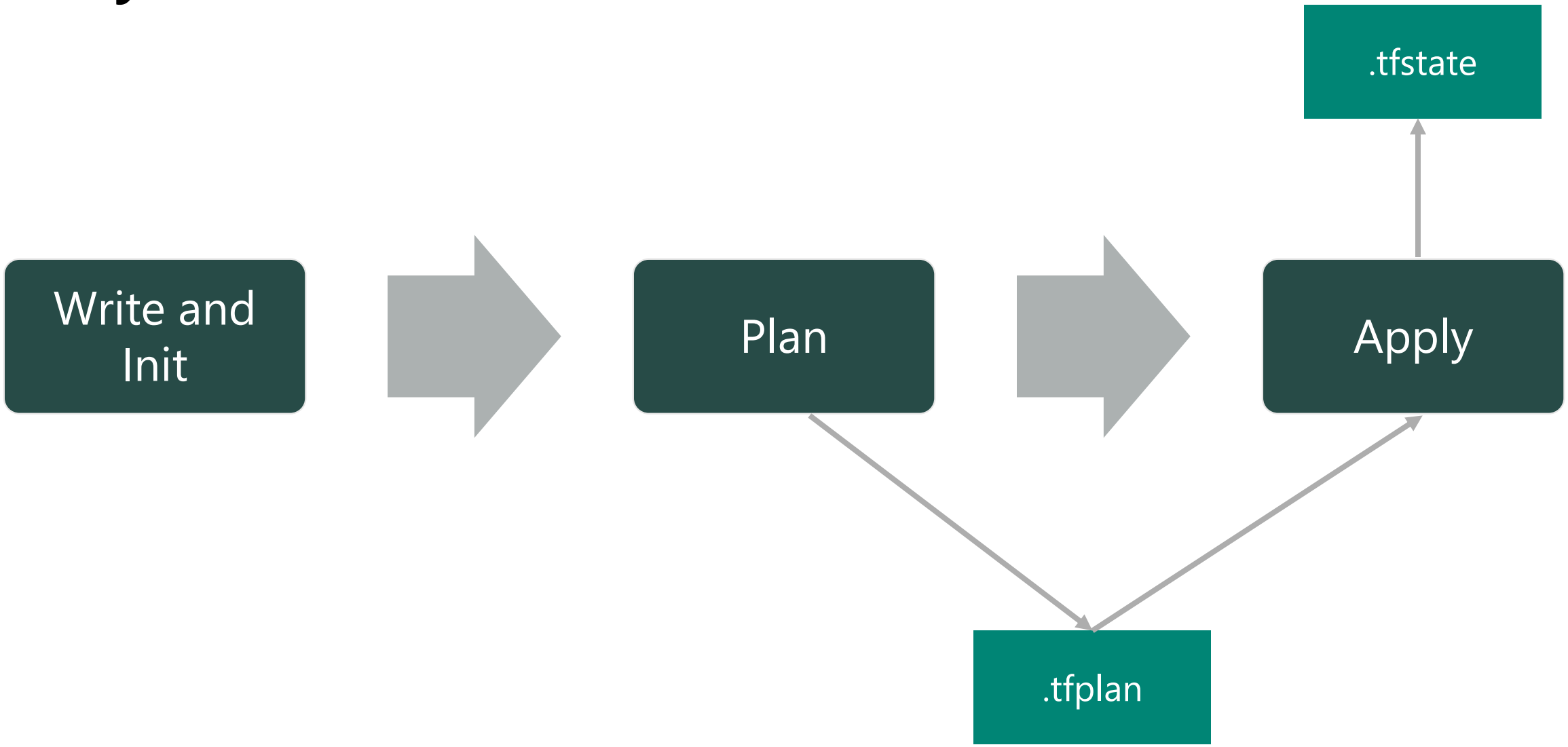
The Core Terraform Workflow



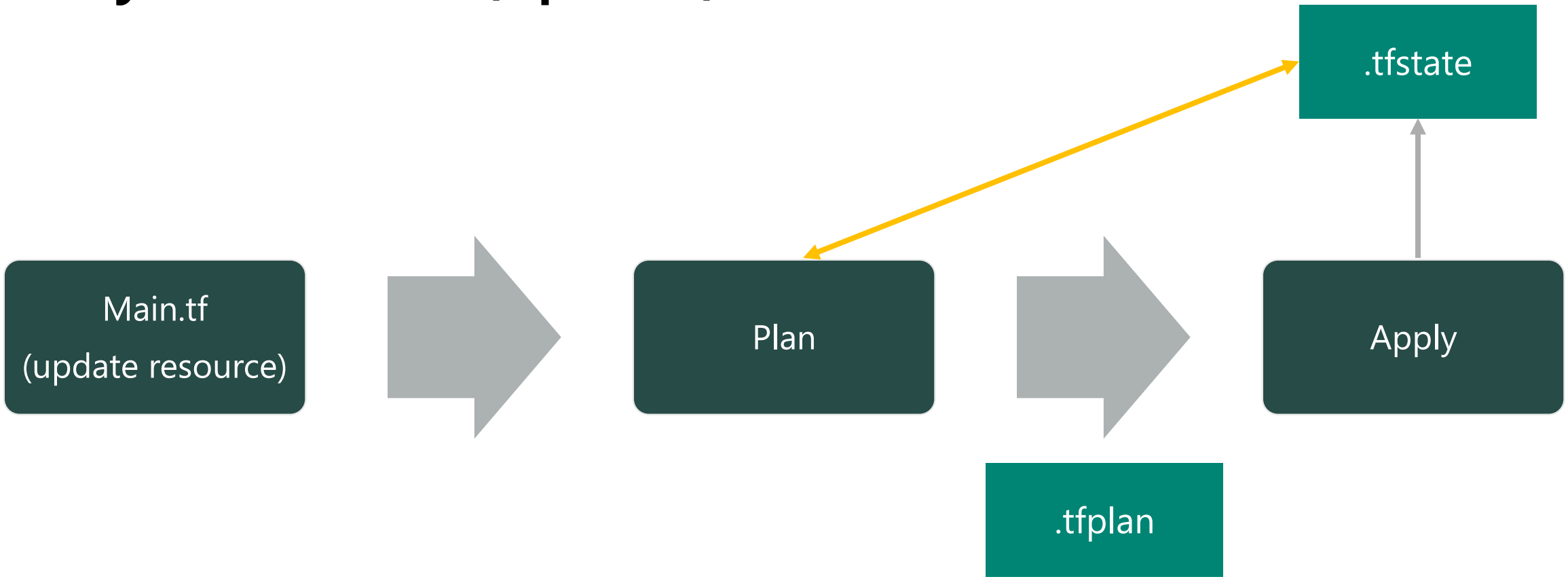
Provisioning with Terraform

Day 0, 1, 2 and N Operations

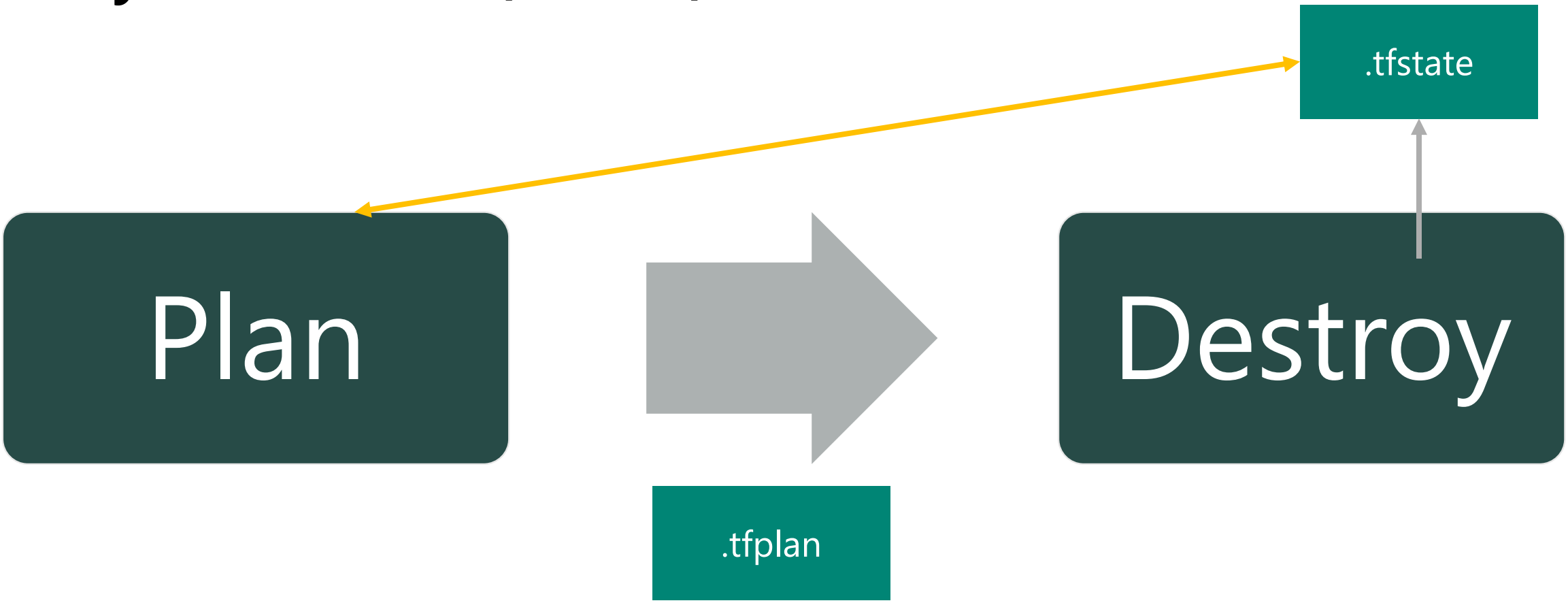
Day 1 Scenario (create)



Day 2 Scenario (update)



Day N Scenario (delete)



Terraform Configuration

Simple Configuration

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "example1" {
  name     = "contoso_rg"
  location = "UK South"
  tags = {
    "cost_center" = "contoso research"
  }
}

resource "azurerm_resource_group" "example2" {
  name     = "contoso_dev_rg"
  location = "UK South"
  tags = {
    "cost_center" = "contoso research"
  }
}
```

Tip: `terraform fmt` to format code according to terraform style guide (ideally after checking in functional changes)

How can we re-use location and tags for both groups?

Local Values

```
locals {
    prefix = "contoso"
    region = "UK South"
    tags = {
        "cost_center" = "contoso research"
    }
}

provider "azurerm" {
    features {}
}

resource "azurerm_resource_group" "example1" {
    name = "${local.prefix}_rg"
    location = local.region
    tags = local.tags
}

resource "azurerm_resource_group" "example2" {
    name = "${local.prefix}_dev_rg"
    location = local.region
    tags = local.tags
}
```

How can we make the region a bit more configurable based on environment?

Variable Definitions

```
# main.tf
variable "region" {
  type = string
  default = "UK South"
}

locals {
    tags = {
        "cost_center" = "contoso research"
    }
}

provider "azurerm" {
   features {}
}

resource "azurerm_resource_group" "example" {
    name = "contoso_rg"
    location = var.region
    tags = local.tags
}
```

```
# terraform.tfvars or auto.tfvars
region = "North Europe"
```

Variable Definitions

```
├── main.tf
├── terraform.tfvars
└── variables.tf
```

```tf
# variables.tf
variable "region" {
  type = string
  default = "UK South“
}
```

```tf
# terraform.tfvars or auto.tfvars
region = "North Europe"
```

```tf
# main.tf
locals {
    tags = {
        "cost_center" = "contoso research"
    }
}

provider "azurerm" {
    features {}
}

resource "azurerm_resource_group" "example" {
    name = "contoso_rg"
    location = var.region
    tags = local.tags
}
```

Environment specific .tfvars *(or .tfvars.json)*

```
├── contoso.europe.tfvars
├── contoso.uk.tfvars
├── main.tf
├── terraform.tfvars
└── variables.tf
```

```
# contoso.europe.tfvars
region = "North Europe"
```

```
# contoso.uk.tfvars
region = "UK West"
```

```
terraform apply -var-file="contoso.uk.tfvars"
```

More Options

```
# Via Command Line

terraform apply –var="region=UK South"
```

The -var option can be used any number of times in a single command

```
# Environment Variables with TF_VAR_ prefix

export TF_VAR_region="UK South"
terraform plan
```

See: https://www.terraform.io/docs/configuration/variables.html

Variable Definition Precedence

Later sources take precedence over earlier ones

- Environment variables
- *terraform.tfvars* file, if present
- *terraform.tfvars.json* file, if present
- Any **.auto.tfvars or *.auto.tfvars.json* files, processed in lexical order of filenames
- Any *var and –var-file* options on the command line, in the order provided

How to return a value?

Output Values
(or just outputs)

```
# main.tf
resource "azurerm_resource_group" "example" {
    name = "contoso_dev_rg"
    location = var.region
    tags = local.tags
}

output "dev_rg_id" {
    value = azurerm_resource_group.example.id
}
```

Expressions

Apply complete! Resources: 2 added, 0 changed, 2 destroyed.

Outputs:

contoso_dev_rg_id = /subscriptions/.../resourceGroups/contoso_dev_rg

```
# show outputs from state file
terraform output
```

Outputs.tf

```
├── contoso.europe.tfvars
├── contoso.uk.tfvars
├── main.tf
├── outputs.tf
├── terraform.tfvars
└── variables.tf
```

```tf
# main.tf
resource "azurerm_resource_group" "example" {
    name = "contoso_dev_rg"
    location = var.region
    tags = local.tags
}
```

```tf
# outputs.tf
output "contoso_rg_id" {
    value = azurerm_resource_group.example.id
    description = "don't show actual data on cli output"
    sensitive = true
}

output "contoso_dev_rg_id" {
    value = azurerm_resource_group.example.id
}
```

Apply complete! Resources: 2 added, 0 changed, 2 destroyed.

Outputs:

contoso_dev_rg_id = /subscriptions/.../resourceGroups/contoso_dev_rg
contoso_rg_id = <sensitive>

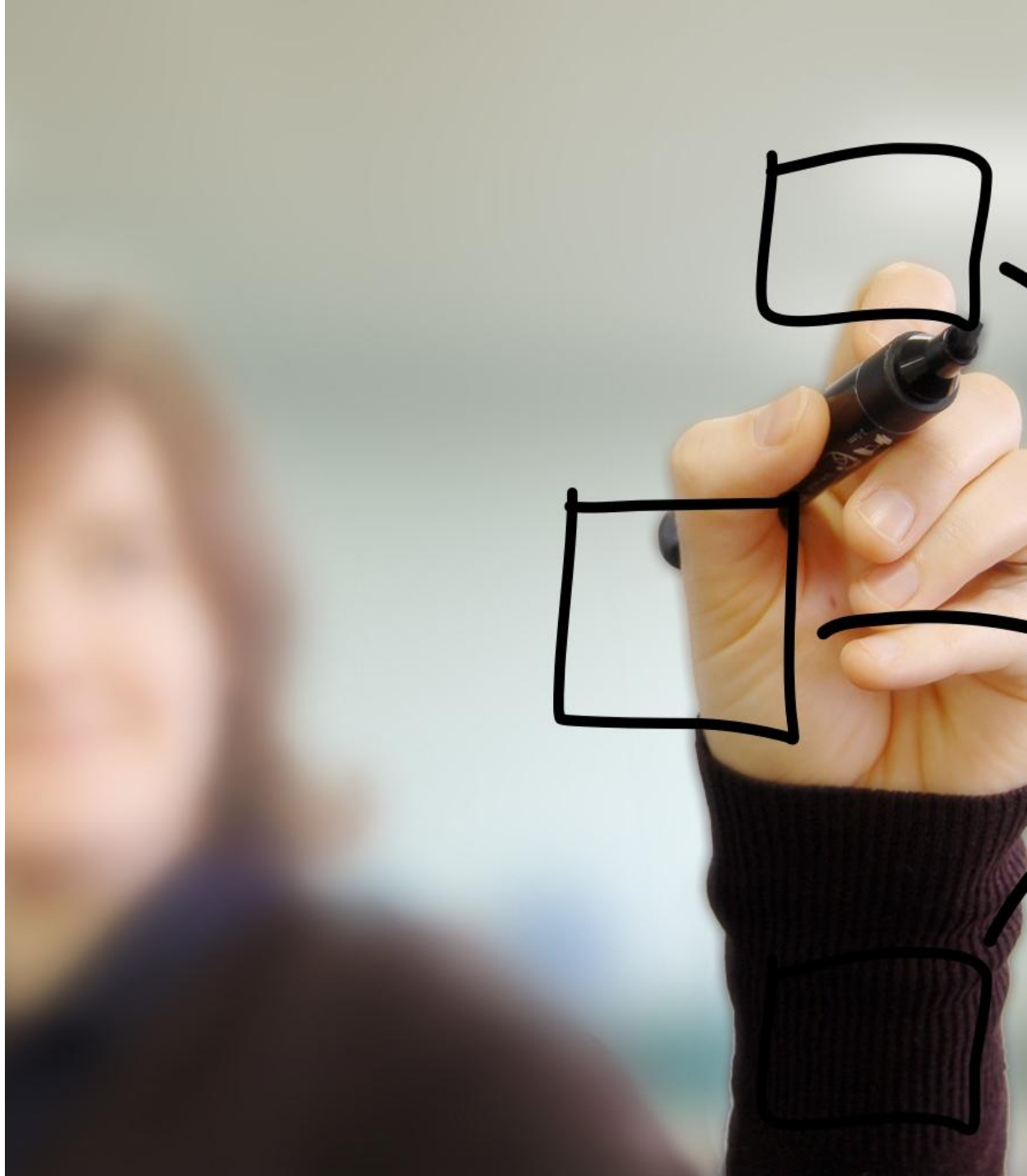
Terraform Basics

Labs 1 and 2

Ask

Discuss

Comment



Microsoft