Infrastructure-as-Code (IaC) with OpenTofu (Terraform)

Hands-on workshop

Patrick Mölk - Feb 4, 2026

Agenda

- Introductions
- Setup prerequisites

1. OpenTofu Basics
2. State management
3. Deploying a simple web app
4. Terragrunt and abstractions
5. Final Exercise

Breaks between chapters
or
as needed

0.1 Intro

Patrick Mölk

Code Smart. Test Hard. Deploy Fast. Build Infrastructure That Lasts.

Cloud Infrastructure, DevOps, CI/CD, Automation, Backend

Freelance IT Consultant and Software Developer

Seven years of professional software development experience. **Let’s connect!**

https://patrick-moelk.eu/

https://linkedin.com/in/patrick-moelk/

https://mastodon.social/@patrick_moelk

Interrupt me!

Download the slides to follow along

I want you to make the most out of this workshop!
I want you to succeed!

Let me know …

- if I wasn’t clear
- if you’re stuck and need help

Raise your hand to get my attention! 🤚

Help each other! You often learn something by explaining things to someone.

Assumptions & Expectations

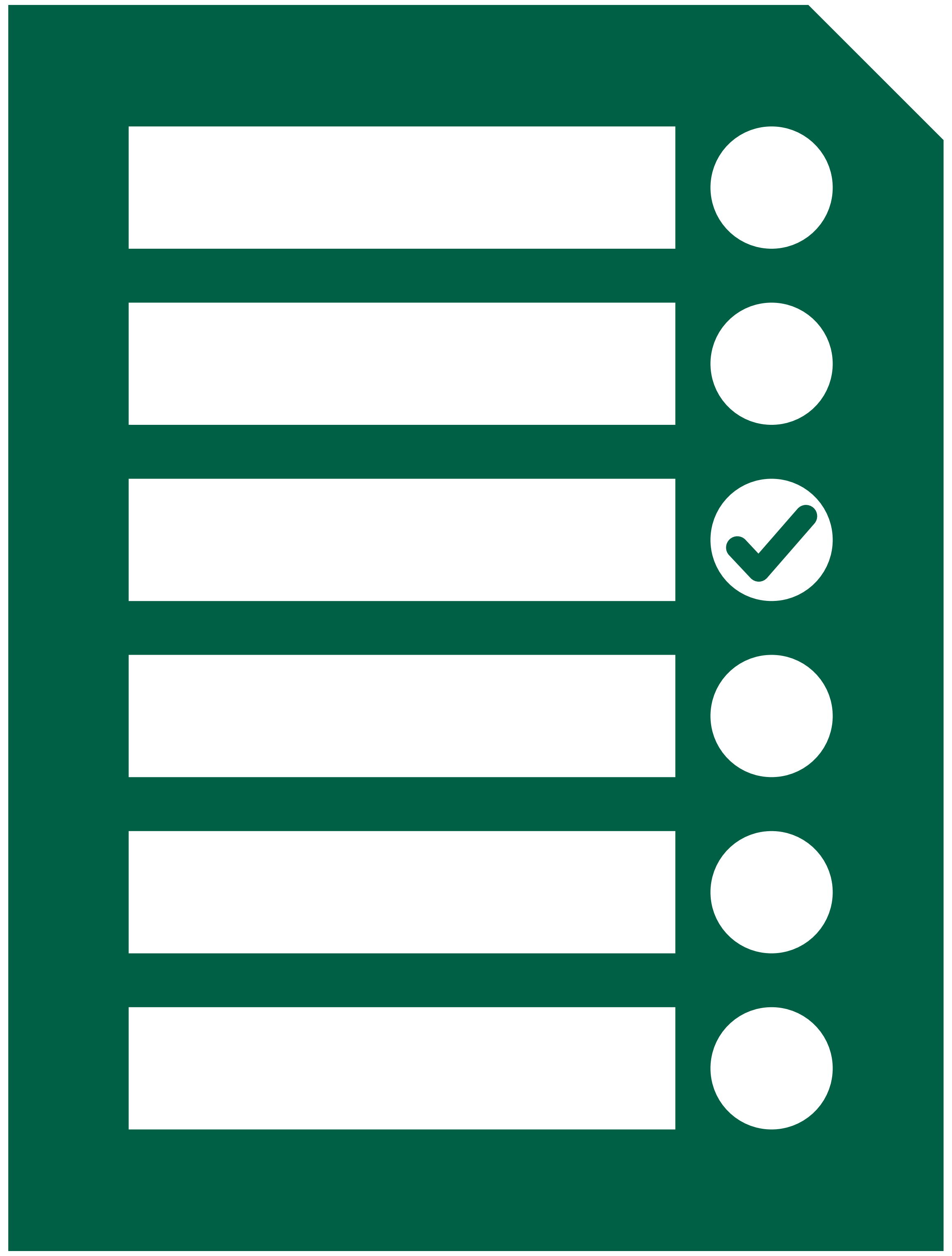
- familiar with basic programming concepts in any language
- have used the command line before
- know of cloud providers and cloud computing in general

- **no knowledge** of OpenTofu / Terraform / Infrastructure-as-Code required
- this is not an architecture workshop
  - we will take a few shortcuts for simplicity to stay focused on IaC
  - I will point out where we're not using best practices: ⚠️🚨

Agenda

- Introductions
- **Setup prerequisites**

1. OpenTofu Basics
2. State management
3. Deploying a simple web app
4. Terragrunt and abstractions
5. Final Exercise



0.2 Prerequisites

Setup Prerequisites

Required

- OpenTofu: https://opentofu.org/docs/intro/install/
  - Homebrew (MacOS and Linux): `brew update; brew install opentofu`
- Terragrunt: https://terragrunt.gruntwork.io/docs/getting-started/install/
  - Hombrew: `brew install terragrunt`
- aws cli: https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html
  - Homebrew: `brew install awscli`
- node / npm (local-exec): https://nodejs.org/en/download
- `git` to clone the workshop repo from either GitLab or GitHub

Setup Prerequisites

Required

- Docker (Desktop) and Docker Compose: https://www.docker.com/products/docker-desktop/

Fake Infrastructure

AWS emulation with localstack

LocalStack

If you do not have access to an actual AWS account

- use the localstack docker container: in the root of the repo run `docker compose up` to spin it up
- in your `/etc/hosts` file add these lines
  - `127.0.0.1          localhost.localstack.cloud`
  - `127.0.0.1          s3.localhost.localstack.cloud`
  - this circumvents a router’s DNS bind protection

Setup Prerequisites

Optional

IDE support for both OpenTofu / Terragrunt is not great! But these extensions help:

- JetBrains IDEs, e.g. PyCharm:
    - https://plugins.jetbrains.com/plugin/7808-terraform-and-hcl
- VSCode:
    - https://marketplace.visualstudio.com/items?itemName=HashiCorp.terraform

Setup Prerequisites

AWS Credentials

- setup `~/.aws/config`

```
[profile opentofu-workshop]
aws_access_key_id = <AWS_ACCESS_KEY_ID>
aws_secret_access_key = <AWS_SECRET_KEY>
aws_account_id = <AWS-account-id>
region = eu-central-1
```

- `export AWS_PROFILE=opentofu-workshop`
- aws tools, like the aws CLI and the OpenTofu provider respect the `AWS_PROFILE` env var

Setup Prerequisites

AWS *Dummy* Credentials

- setup `~/.aws/config` for localstack users with *dummy* credentials

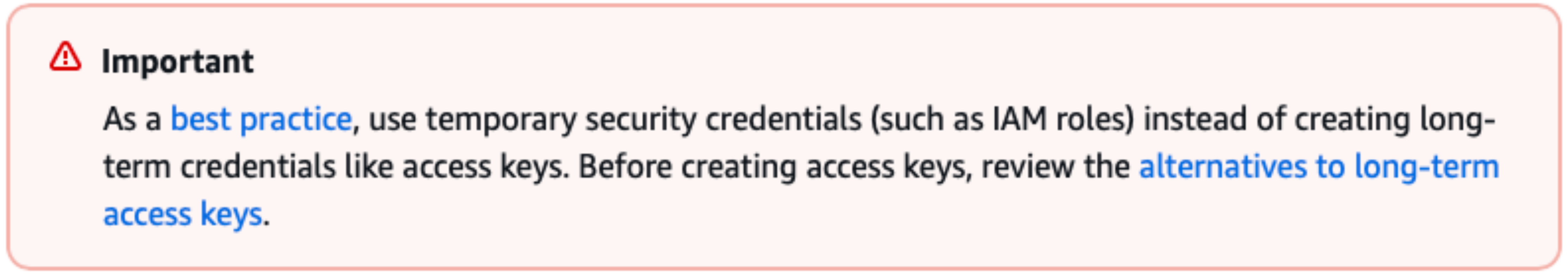
```
[profile opentofu-workshop]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = 1234567890123456789012345678901234567890
aws_account_id = 123456789012
region = eu-central-1
```

- `export AWS_PROFILE=opentofu-workshop`
- aws tools, like the aws CLI and the OpenTofu provider respect the `AWS_PROFILE` env var

Setup Prerequisites

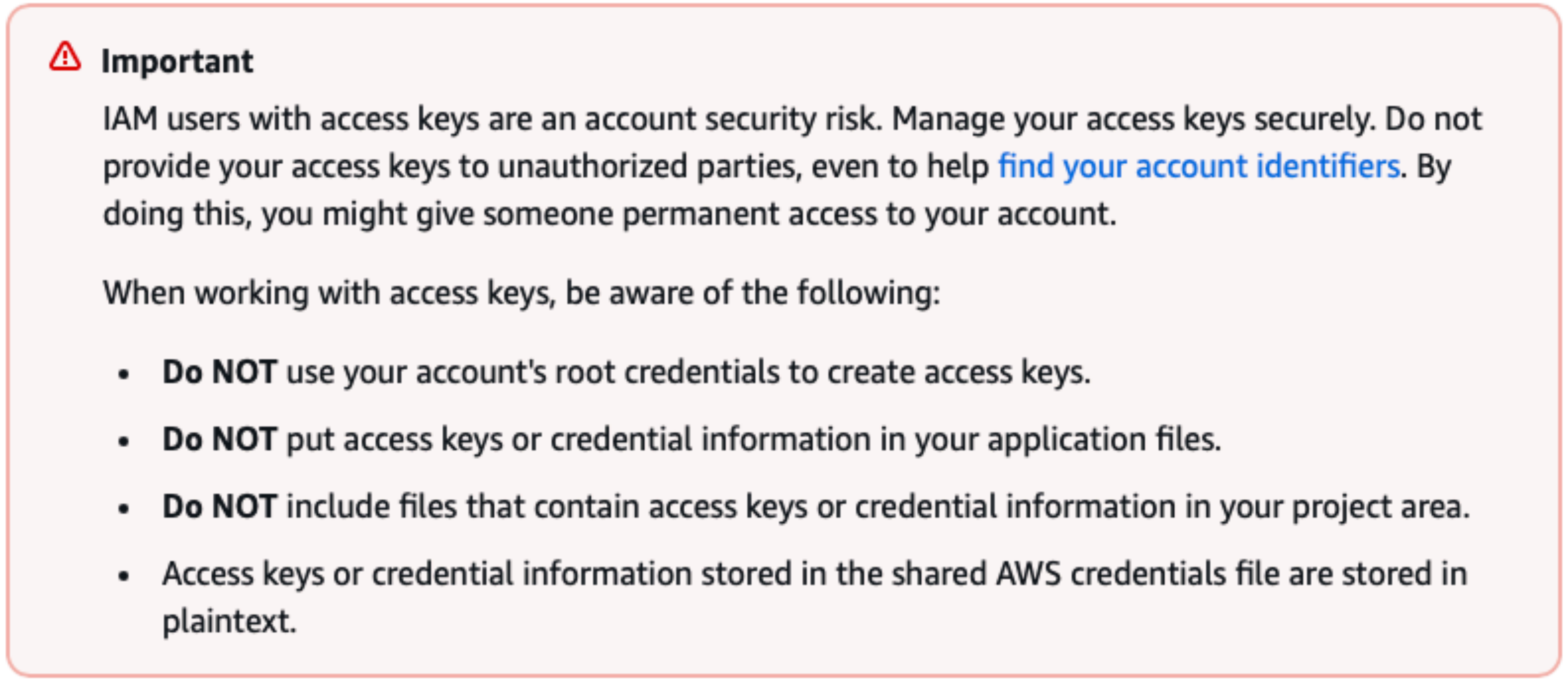
Best practice: Side note on AWS credentials

- consider “access key id” and “secret access key” to be deprecated / unsafe
- use temporary credentials instead, e.g. with **AWS Identity Center** (AWS SSO)
- we will use “access key id” and “secret access key” for simplicity
  ⚠️ DO NOT USE THEM IN PRODUCTION! ⚠️



⚠ **Important**

As a best practice, use temporary security credentials (such as IAM roles) instead of creating long-term credentials like access keys. Before creating access keys, review the alternatives to long-term access keys.



⚠ **Important**

IAM users with access keys are an account security risk. Manage your access keys securely. Do not provide your access keys to unauthorized parties, even to help find your account identifiers. By doing this, you might give someone permanent access to your account.

When working with access keys, be aware of the following:

- **Do NOT** use your account's root credentials to create access keys.
- **Do NOT** put access keys or credential information in your application files.
- **Do NOT** include files that contain access keys or credential information in your project area.
- Access keys or credential information stored in the shared AWS credentials file are stored in plaintext.

read more in AWS docs and
on this and the following slides

Agenda

- Introductions
- Setup prerequisites

1. **OpenTofu Basics**
2. State management
3. Deploying a simple web app
4. Terragrunt and abstractions
5. Final Exercise

1. OpenTofu Basics

1. OpenTofu Basics

Agenda

- OpenTofu origin story and language
- Providers and resources
- Variables, locals, and outputs
- Destroying infrastructure

1.0 OpenTofu Origins

A story of true open source

- **OpenTofu** is a fork of **Terraform** (by HashiCorp)
- In 2023 HashiCorp changed the Terraform license
- New license is no longer truly open source (BUSL: Business Source License)
- OpenTofu was forked to be truly open source (MPL-2.0: Mozilla Public License v2.0)
- OpenTofu is since developed independently by a newly founded foundation
- OpenTofu aims to be a drop-in replacement for Terraform
  - `alias tf=terraform` (old)
  - `alias tf=tofu` (new)
- Read more in their blog post

1.0 OpenTofu Language

- declarative language
  - describes desired end state
- not imperative, not step-by-step instructions
- implicit and explicit relationships between resources may dictate order of operations
- HCL syntax (HashiCorp Configuration Langauge)
  - similar to JSON, but easier for humans to read
- all `.tf` files in a folder (“module”) are considered, there is no `main()` entry point

OpenTofu

1.1 Providers and Resources

Providers

- plugins enabling us to interact with cloud providers and other APIs
- think of providers like libraries in other programming languages
- providers often need to be configured
  - API Keys, cloud regions, endpoint URLs
- providers add resource types and / or data sources
  - think: classes and functions that a library exposes to developers

read more

OpenTofu

1.1 Providers and Resources

Resources

- Resources represent the pieces of infrastructure that we want to manage
- examples:
  - virtual machines, compute instances, storage, databases
  - virtual networks, firewall settings
  - DNS records

read more

OpenTofu

OpenTofu Language

Example: provision an S3 bucket in AWS

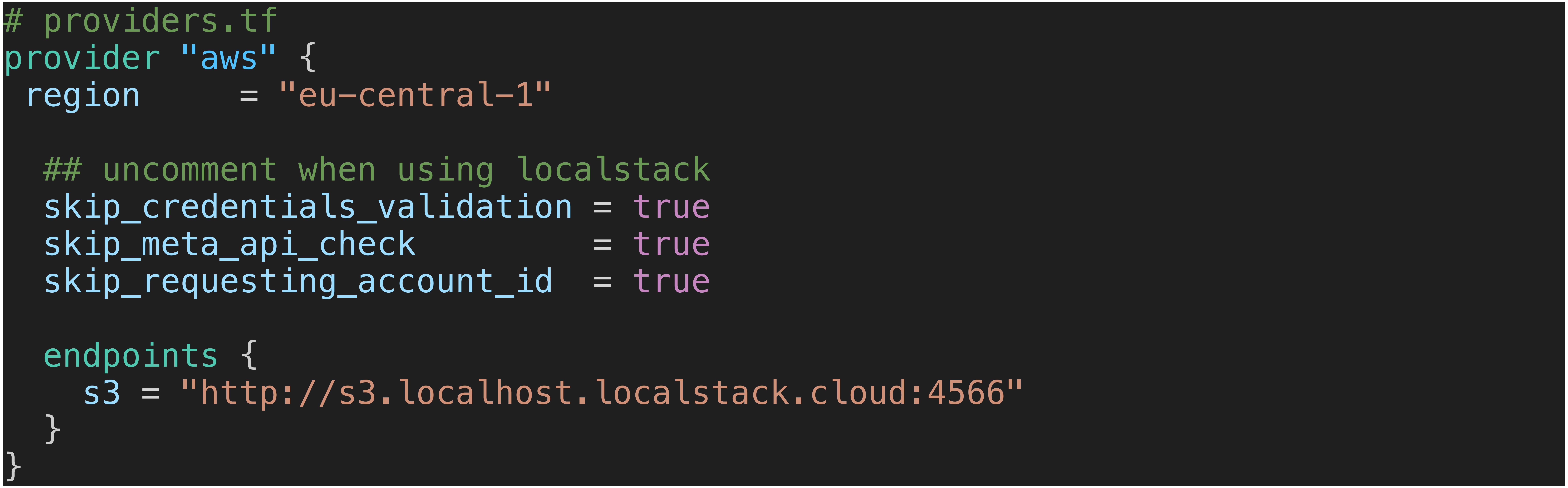
```
# providers.tf
provider "aws" {
 region      = "eu-central-1"
}


resource "aws_s3_bucket" "my_bucket" {
  bucket = "my-bucket"
}
```

- the AWS provider also respects the `AWS_PROFILE` env var and will pull the credentials from `~/.aws/config` accordingly

OpenTofu Language

AWS provider with localstack and dummy credentials



```
# providers.tf
provider "aws" {
 region      = "eu-central-1"

  ## uncomment when using localstack
  skip_credentials_validation = true
  skip_meta_api_check         = true
  skip_requesting_account_id  = true

  endpoints {
    s3 = "http://s3.localhost.localstack.cloud:4566"
  }
}
```

- make sure localstack is running: run `docker compose up` at root of repo

01 Basics - Exercise 01

Providers and resources

Let’s have a look at the code for the first exercise
in `01-basics/01-providers-and-resources/`

- in `providers.tf`
  - a `terrform {}` block lists required providers and versions
  - a `provider "<provider>" {}` block configures a provider

- in `s3.tf` we define a bucket with its name and default values
  - replace `YOUR-NAME` to satisfy the global uniqueness constraint

01 Basics - Exercise 01

Providers and resources

Let's have a look at the code for the first exercise
in `01-basics/01-providers-and-resources/`

- in `providers.tf`
  - a `terrform {}` block lists required providers and versions
  - a `provider "<provider>" {}` block configures a provider
    - uncomment the localstack specific lines
- in `s3.tf` we define a bucket with its name and default values
  - replace `YOUR-NAME` to satisfy the global uniqueness constraint

01 Basics - Exercise 01 cont’d

Providers and resources

- open a terminal in the workshop repo directory `01-basics/01-providers-and-resources`
- run `tf init` (assumption: `alias tf=tofu`)
- run `tf apply`
  - take a look at the planned changes, then type `'yes'` to approve and press ENTER
  - address any errors, re-run `tf apply`
  - you should see something like this:

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

- congrats you deployed something with OpenTofu 🥳

1.2 Variables

Cleanup 🧹

Let’s avoid **hard coded values** (and magic numbers) with variables

- we’ll define variables for
  - region
  - bucket name

1.2 Variables

Example definition and usage

```
# definition
variable "aws_region" {
  type        = string
  description = "AWS region, e.g. eu-central-1 or us-east-1"
  default     = "eu-central-1"
}

variable "s3_bucket_name" {
  type        = string
  description = "name for S3 bucket"
}

# usage
provider "aws" {
 region = var.aws_region
}

resource "aws_s3_bucket" "my_bucket" {
  bucket = var.s3_bucket_name
}
```

1.2 Variables

- enable us to easily change values in a single place
- avoid hard coding values, including secrets
- essential for reusable modules (see chapter 3)

OpenTofu

1.2 Variables

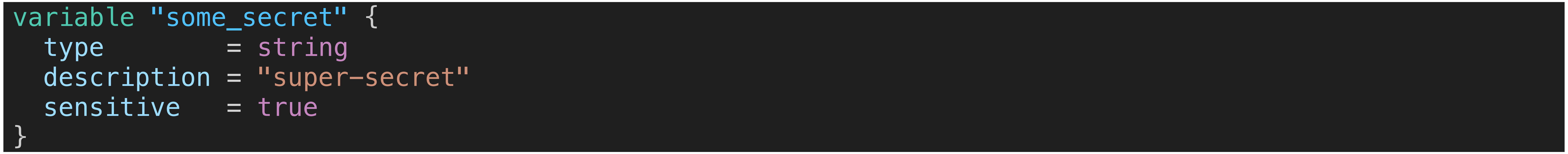
Defaults

- setting default values is not necessarily bad practice
  - it's similar to a constant in other programming contexts
- with no default value, we're prompted to set a value when executing `tf apply`
  - useful for secrets that MUST be outside of version control … but there are better solutions

OpenTofu

1.2 Variables

Setting variables’ values and assigning types



```
variable "some_secret" {
  type        = string
  description = "super-secret"
  sensitive   = true
}
```

- variables can be overwritten or set via environment variables: `TF_VAR_<variable-name>`, e.g. `export TF_VAR_some_secret=supersecret; tf apply`

- set type, description, sensitive (boolean, for secrets), and validation condition
  - primitive types: `string`, `number`, `bool`
  - complex types: `list(<TYPE>)`, `set(<TYPE>)`, `object()`, and more

Exercises

General rule

- for most(!) (not all) exercises you will build on top of the solution of the previous exercise
    - example: Chapter 01 - Exercise **02**
    - working directory: 01-basics/*01*-providers-and-resources
    - solution directory: 01-basics/**02**-variables-and-outputs
- The first line in each exercise will tell you the **working directory**
    - generally for exercise X you will **work** in directory *X–1* of the chapter
    - the **solution** for exercise X will be in directory X of the chapter

01 Basics - Exercise 02

Variables

- you’ll modify `01-basics/01-providers-and-resources/`
- your final version of `01-basics/01-providers-and-resources/` should be similar to `01-basics/02-variables/` after the exercise
- feel free to peek at the solution if you’re stuck or to draw inspirations, but
  - I encourage you to find your **own solutions** based on the previous slides
  - download the slides so you can go back and forth!

01 Basics - Exercise 02

create and use variables

in `01-basics/01-providers-and-resources`

1. create a `variables.tf` file to define variables (e.g. bucket, region)
   - use these variables in `s3.tf` and `providers.tf`
   - in your terminal run `tf apply`
2. play around: see what happens when you …
   - set or remove `default` values and `description` or
   - toggle the boolean `sensitive` flag (for variables without default values)

1.2 Variables

Side note: Secrets & keeping them out of version control

I typically use a `.env` file (globally `.gitignore`d!)

```
# .env
export AWS_PROFILE="opentofu-workshop"
export TF_VAR_db_password="secret"
```

- usage: `source .env; tf apply`
- convenient solution, when working with multiple AWS accounts / profiles
- pros: secrets not in version control, AWS secrets especially safe 🔒
- cons: other secrets (DB, API Keys, etc.) in plain text
- better solutions (not covered here): AWS Secrets Manager, HashiCorp Vault

1.3 Locals

Definition and usage

- similar to variables, but can store results of expressions
- internal, cannot be set from the outside via prompts or `TF_VAR_...`
- example: string-interpolation with variables

```
locals {
  bucket_name = "${var.s3_bucket_name}-${var.name}"
}

resource "aws_s3_bucket" "my_bucket" {
  bucket = local.bucket_name
}
```

01 Basics - Exercise 03

local variables and expressions

in `01-basics/02-variables/`

- add a new variable `name`
- create a local variable to create a dynamic bucket name based on `var.name` and `var.s3_bucket_name` (or however you named your variables)
- set the bucket name in `s3.tf` using the local variable
- run `tf apply`

Feel free to continue with **bonus exercises** on your own

01 Basics - Exercise 03.1

local variables and expressions - Bonus 1 🌟

in `01-basics/02-variables/`

- add a new variable `environment`
  - use it as a suffix in `local.bucket_name`
- the bucket names should be `<bucket_name>-dev`,`<bucket_name>-staging`, BUT just `<bucket_name>` when `var.environment == "prod"`

- hints: in OpenTofu there…
  - is a tertiary operator: `<condition> ? <then> : <else>`
  - are logical operators: `&&`, `||`, `==`

01 Basics - Exercise 03.2

local variables and expressions - Bonus 2 🌟

in `01-basics/02-variables`

- make sure `var.environment` can **only** be `dev`, `staging`, or `prod`
- remove any default value for `var.environment`
- run `tf apply` multiple times with valid and invalid values for `environment`

hint: use a nested validation block inside the `variable` block

1.4 Outputs

- modules can produce outputs
- outputs can be passed as inputs to other resources and modules
- print useful information like a domain or IP address
- resources contain properties that can be accessed with the dot-notation

```
output "hello" {
  value = "world"
}

output "s3_bucket_domain" {
  value = aws_s3_bucket.my_bucket.bucket_domain_name
}
```

01 Basics - Exercise 04

Outputs

in `01-basics/03-locals`

1. create an `outputs.tf` file and define outputs in it
   - run `tf apply` and take notice of the output values
2. play around, e.g.
   - see what happens when you set an entire resource as the output

1.5 Destroying infrastructure 💣💥

- command: `tf destroy`
- removes all infrastructure resources managed by OpenTofu
- ⚠️ Use with caution, data loss may occur ⚠️

💣💥

01 Basics - Exercise 05

Destroying infrastructure

in `01-basics/04-outputs`

- run `tf init; tf apply`
- run `tf destroy`
  - double check what would be destroyed
  - type `'yes'` and hit ENTER to approve

If your fast, feel free to skip ahead and do bonus exercises. 💣💥

1.5 Destroying infrastructure

Data loss!

- ⚠️ Data loss may occur! ⚠️
- some resources, e.g. RDS and EC2 instances, can be destroyed even with data
  - create a backup before destroying OR
    be ABSOLUTELY sure you do not need the data
- other resources, like S3, cannot be destroyed when they contain data

💣💥

1.5 Destroying infrastructure

Data loss! Bonus exercises

- use the shell scripts in `<repo-root>/scripts/` to upload and delete files
  - `upload-file-to-s3.sh <bucket> <local-file-path> <s3-key>`
  - `delete-key-in-s3.sh <bucket> <s3-key>`

💣💥

BONUS

01 Basics - Exercise 05.1

Destroying infrastructure - Bonus 1 🌟

in `01-basics/04-outputs`

- run `tf apply`

- upload a file to your bucket (script, see previous slide)
- run `tf destroy` and see what happens

- delete the file from your bucket (script, see previous slide)
- run `tf destroy` again 💣💥

01 Basics - Exercise 05.1

Destroying infrastructure - Bonus 1 🌟

in `01-basics/04-outputs`

- run `tf apply`
- `export AWS_ENDPOINT_URL=http://localhost:4566`
- upload a file to your bucket (script, see previous slide)
- run `tf destroy` and see what happens

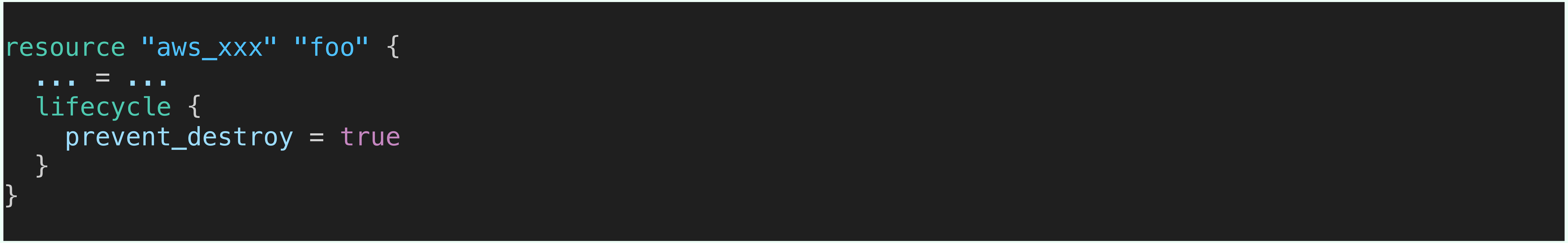
- delete the file from your bucket (script, see previous slide)
- run `tf destroy` again

💣💥

1.5 Destroying infrastructure

Safeguard

- a `lifecycle` block can safe-guard against accidental destruction
- useful for resources that can be deleted even when containing data



```
resource "aws_xxx" "foo" {
  ... = ...
  lifecycle {
    prevent_destroy = true
  }
}
```

BONUS

01 Basics - Exercise 05.2

Destroying infrastructure - Bonus 2 🌟

in `01-basics/04-outputs`

- add a lifecycle block with `prevent_destroy = true`
- run `tf apply`
- run `tf destroy` and see what happens
- set `prevent_destroy = false`, then run `tf destroy` again

- optional cleanup: go back to previous exercises and destroy the buckets
  - `aws s3 ls` to list buckets

💣💥

01 Basics - Exercise 05.2

Destroying infrastructure - Bonus 2 🌟

in `01-basics/04-outputs`

- add a lifecycle block with `prevent_destroy = true`
- run `tf apply`
- run `tf destroy` and see what happens
- set `prevent_destroy = false`, then run `tf destroy` again

- optional cleanup: go back to previous exercises and destroy the buckets
  - `export AWS_ENDPOINT_URL=http://localhost:4566`
  - `aws s3 ls` to list buckets

💣💥

1. OpenTofu Basics

Wrap up

- providers and resources, `tf apply`
- variables, locals, outputs
- keeping secrets safe and out of version control
  - best practice: use temporary credentials and secret managers
- `tf destroy`
  - data loss may occur!

- **reminder**: run `tf destroy` to save costs!

OpenTofu

1. OpenTofu Basics

Resources

- Tofu: Providers
- Tofu: Resources
- Tofu: Variables, Locals, Outputs
- Tofu: Variable Custom Validation Rules
- `tf apply`
- `tf destroy`
- AWS Access Keys
- AWS SSO setup

Break?

Agenda

- Introductions
- Setup prerequisites

1. OpenTofu Basics
2. **State management**
3. Deploying a simple web app
4. Terragrunt and abstractions
5. Final Exercise

2. State Management

2. State Management

Agenda

- What is state?
- Working with state: `tf state` commands
- Manipulating state: Renaming a resource
- Manipulating state: Importing existing infrastructure
- Remote state to enable collaboration (with locking)
  - State storage backends: S3, GitLab
  - Bootstrapping problem

2.0 What is state?

State Management

- “database” to map resources in configuration to real world resources
  - e.g. `resource "aws_s3_bucket" "my_bucket"` maps to an actual S3 bucket identified by its name
- keeps track of managed infrastructure and its metadata
- needed to determine which changes to make to infrastructure
- detects changes made outside of code (drift)
- by default stored in a local `terraform.tfstate` file in JSON format
- read more about state and its purpose

2.0 What is state?

State Management

configuration (`.tf` files)	**<- diff ->**	**state**	**<- diff ->**	**real world** (e.g. cloud resources)
what you want to exist		what should actually exist		what does actually exist
	something needs to be created, modified or deleted (destroyed)		DRIFT WARNING: someone or something other than tf changed some piece of infra!	

2.1 Working with State

State Management

- base command `tf state`
- `tf state list` shows all resources’ addresses
- `tf state show` prints state data for a specific resource
- `tf state mv` renames a resource
- `tf state rm` stops managing a resource without destroying it
- `tf import` imports an existing resource to be managed with OpenTofu

2.1 Working with State

State Management: renaming / moving a resource

- `tf state mv <old-address> <new-address>`
- address contains resource type and name, example:

```
tf state mv aws_s3_bucket.foo aws_s3_bucket.bar
```

- find resource addresses with
  - `tf state list` or
  - `tf state ls` if you’re lazy like me

02 State - Exercises

Reminder

If you're using localstack

- uncomment lines in `providers.tf`!
- this will be required every time we run an exercise in a new directory!

02 State - Exercise 01

Renaming a resource

in `02-state/01-02-rename-and-rm-resource`

- run `tf init; tf apply`
- I have introduced a typo in one of the resource names / addresses
- run `tf state ls` to find the typo
- fix the typo in all affected `.tf` files
- run `tf apply` and **take a close look** at the changes!
  Should you apply them?
- use `tf state mv` to rename the resource in state to match the `.tf` file
- run `tf apply`, there should be no changes

2.1.1 Data sources

Did you notice… ?

- … that you didn’t need to change the bucket name?
  - I snuck in a **data source** running `whoami` on your machine to get your current user name, see `s3.tf`

In Chapter 1, I left something out

- providers also add something called **data sources**
- rather than creating resources, data sources get, fetch or create data on the fly
  - often from your cloud provider, e.g. Amazon Machine Images (AMIs)

read more about data sources

2.2 Removing resources from state

State Management

- we may want to stop managing pieces of infrastructure with OpenTofu
  - e.g. when we want to manage it in another OpenTofu repo
- but we do NOT want to destroy it!
- command `tf state rm <resource-address>`

02 State - Exercise 02

Removing resources from state

in `02-state/01-02-rename-and-rm-resource`

- run `tf state rm aws_s3_bucket.my_bucket`
- verify that the bucket is no longer in state with `tf state ls`
- verify that the bucket is not deleted, e.g. with `aws s3 ls`
- what happens when you run `tf apply -auto-approve` now?
  - take a look at the state, what do you see?

usually we would now remove the resource from our configuration…

2.3 Importing existing infrastructure

State Management

- … but this is a perfect setup for the next exercise
- imagine we created the resource by hand in the cloud provider’s web UI
- now we want to manage it in OpenTofu
- use `tf import <resource-address> <resource-id>`
  - the resource IDs are fairly different between types of resources
  - often ARNs for AWS resources, but only “bucket name” for S3
  - consult the documentation for the resource at the very bottom

02 State - Exercise 03.1

Importing existing resources into state

in `02-state/03-import-resource`

- run `tf init`
- run `tf import aws_s3_bucket.my_bucket <bucket-name>`
  - bucket name: `iac-workshop-bucket-02-<whoami>`
- run `tf state ls` to verify that the bucket is imported
- what do you expect to happen when you run `tf apply` now?

- to save costs: delete files in the bucket and run `tf destroy`

Feel free to continue with bonus content and exercises on the following slides

2.3 Importing existing infrastructure

State Management

- there's another way to import existing resources
- using an `import` block:
  - `to`: resource's address in state
  - `id`: resource's id
  - in our case

```
import {
  to = aws_s3_bucket.my_bucket
  id = local.bucket_name
}
```

- the import block can be deleted after successful import with `tf apply`

BONUS

02 State - Exercise 03.2

Importing existing resources into state - Bonus 🌟

in `02-state/03-import-resource`

- run `tf state rm aws_s3_bucket.my_bucket`
- add an `import` block for the S3 bucket
- run `tf apply`
  - take a close look at the planned changes before approving
  - notice anything?
- verify that the bucket is in state now with `tf state ls`

- to save costs run `tf destroy`

2.4 Generate code from existing infra

State Management - Bonus

- OpenTofu can generate the configuration for existing resources
- useful for complex resource configurations, but be careful ⚠️

> **[OpenTofu] produces HCL to act as a template that contains [OpenTofu]’s best guess at the appropriate value for each resource argument.**

- use an `import` block like before
- run `tf plan -generate-config-out=“generated.tf”`
- ⚠️ the feature is still considered **experimental** ⚠️

BONUS

02 State - Exercise 04

Generating config for existing resources - Bonus 🌟

in `02-state/04-generate-config`

- run `tf init`
- add an `import` block in any `.tf` file
  - bucket name: `iac-workshop-bucket-02-<whoami>`
- run `tf plan -generate-config-out="s3.tf"`
- compare `s3.tf` with the S3 configuration from previous exercises
- adapt the generated file if necessary or desired
- run `tf apply`, optional: remove the `import {}` block
- to save costs run `tf destroy`

2.5 Remote State

State Management

- Enables multiple people & automation (CI/CD) to manage same infrastructure
- Prevents simultaneous modifications via state locking, if enabled
- State storage backends:
  - local (default)
  - S3
  - http (REST)
  - and more

2.5 Remote State

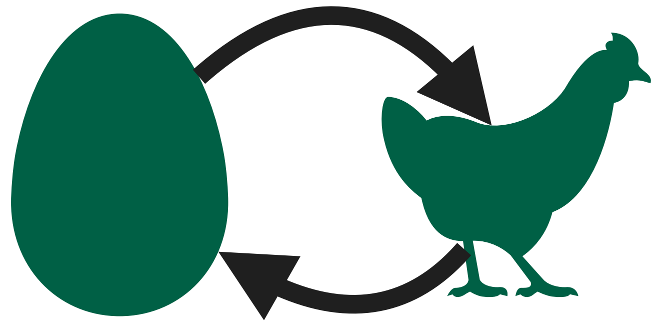
State Management — bootstrapping problem

Our goals

- provision main infrastructure in code, e.g. an EC2 machine
- enable collaboration: store the OpenTofu state remotely, e.g. in S3

How do we provision the S3 bucket for the remote state?

- This is called the **bootstrapping problem** (“chicken egg problem”)
  - we need state to create infrastructure
  - we need infrastructure to store the state

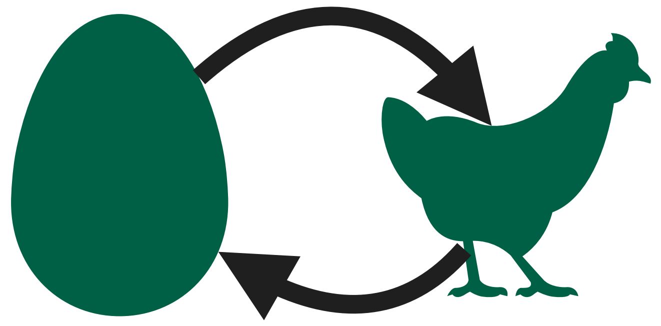


2.5 Remote State

Solutions to the bootstrapping problem

In order of (my) preference

1. use a service other than your cloud provider to store state (avoid the problem)
    - e.g. GitLab via the “http” backend
2. use terragrunt (more on that in chapters 4)
3. **hacky** (exercise)
    1. provision the S3 bucket for the remote state with local state first
    2. then move the local state to the S3 bucket
4. “click-ops”: setup the remote state backend manually, not in code
    - i.e. create S3 bucket for the state via AWS console by clicking around



2.5 Remote State

S3 Backend

```
# providers.tf
terraform {
  # ...
  backend "s3" {
    region       = "eu-central-1"
    bucket       = "<bucket-name>"
    key          = "<key>" # e.g. tofu.tfstate
    encrypt      = true
    use_lockfile = true

    ## uncomment when using localstack
    # endpoints = {
    #   s3 = "http://s3.localhost.localstack.cloud:4566"
    # }
  }
}
```

02 State - Exercise 05.1

Remote state

in `02-state/04-generate-config`

- from a previous exercise copy `s3.tf` (if you didn’t run the Bonus Exercise 04)
- add a `state.tf` file with another S3 bucket config
  - e.g. copy & paste `s3.tf`, change the bucket name and resource address
- run `tf apply` to create the two buckets (one “regular”, one for state)
- in `providers.tf` nest a [`backend "s3"`]() block in the `terraform` block
  - set at least the arguments: `region`, `bucket`, `key`, and `use_lockfile`
- run `tf init -migrate-state`, enter `'yes'` to approve when prompted
- run `tf apply`

02 State - Exercise 05.2

Remote state

This exercise demonstrates the locking mechanism. It emulates two people trying to modify infrastructure simultaneously.

In `02-state/04-generate-config`

- in two terminals run `tf apply` as simultaneously as possible
  - e.g. if you use iTerm2:
    - open a terminal, press `CMD + D` to split the window into two panes
    - press `CMD + OPTION + i` to type in both panes simultaneously
    - type `tf apply` and press ENTER to run the command simultaneously

02 State - Exercise 05.3

Bonus/Optional cleanup (save costs)

in `02-state/04-generate-config`

- run `tf destroy`, what do you expect to happen?
- how can we delete the bucket storing our state?
- comment out the `backend "s3" {}` block
- run `tf init -migrate-state`
- delete all files in all managed buckets
- run `tf destroy`

2. State management

Wrap up

- state represents what should currently be deployed
- `tf state` commands to list resources, read metadata, etc.
- manipulate state:
  - rename resources
  - import existing infra into state
- remote state with locking for safe collaboration

- **reminder**: save costs and run `tf destroy`
  - execute `<root>/scripts/empty-and-delete-s3-bucket.sh <bucket>`

2. State management

Resources

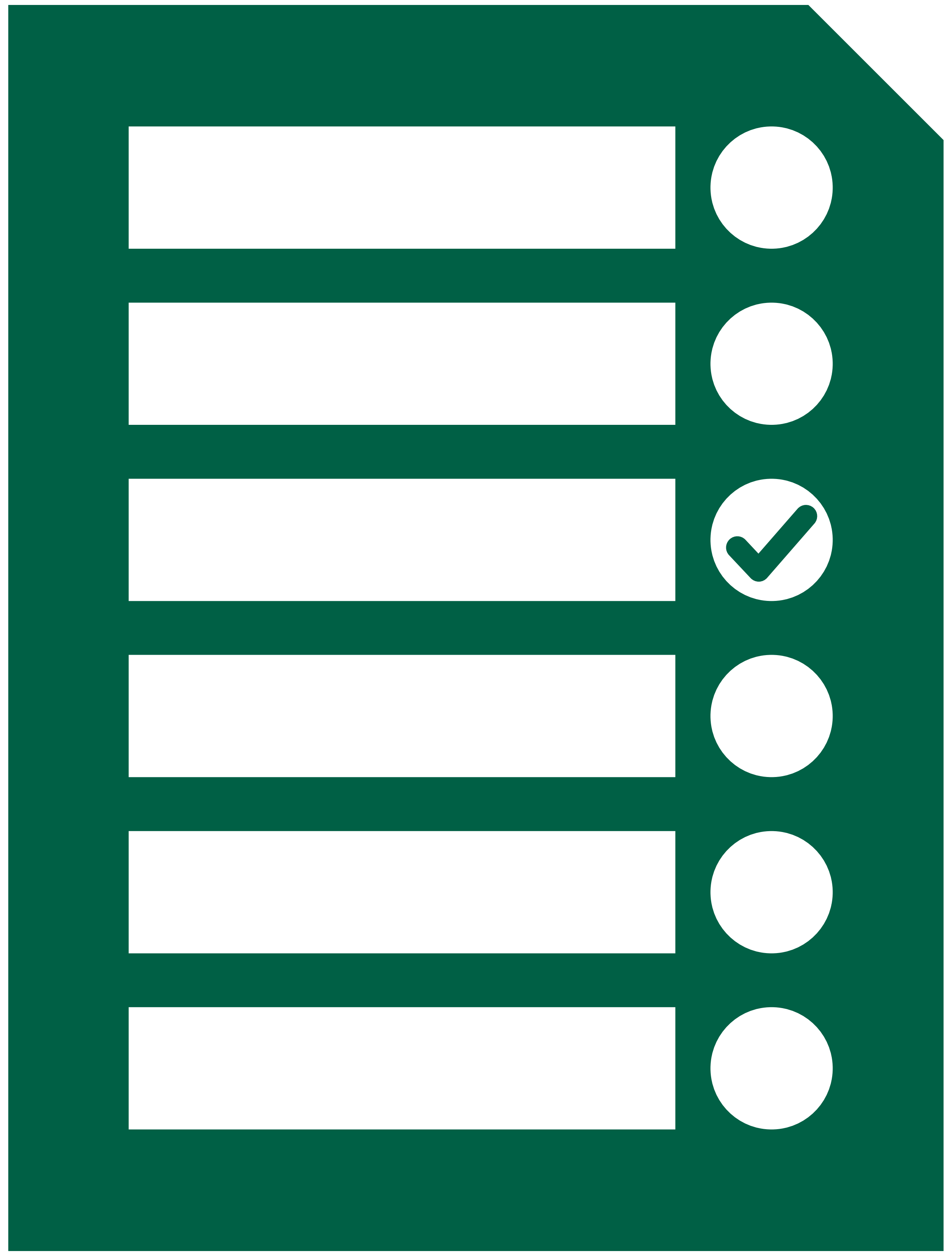
- State and its Purpose
- Data Sources
- `tf state`
- `tf import`
- Generating Configuration
- Backend Configuration

Break?

Agenda

- Introductions
- Setup prerequisites

1. OpenTofu Basics
2. State management
3. **Abstractions and Terragrunt**
4. Deploying a simple web app
5. Final Exercise



3. Abstractions and Terragrunt

3. Abstractions and Terragrunt

Agenda

- OpenTofu modules
- What’s Terragrunt and why use it?
- Terragrunt abstraction layers
    - units
    - stacks
- Solution to the bootstrapping problem

3.1 Abstractions

OpenTofu Modules

- a directory containing `.tf` files is a **module**
- every directory in which we have executed `tf apply` so far is a **module**
- modules are reusable
- in Object Oriented Programming (OOP) terms:
  - modules are classes which can be used or “instantiated” multiple times
  - a module can be an instance of itself when running `tf apply` in it

OpenTofu

3.1 Abstractions

OpenTofu Modules

- importing a module in a `.tf` file

```
module "file" {
  source = "../../module"

  input_variable = var.foo
}
```

- `source` can be
  - a relative path to a local module
  - a registered module like `"terraform-aws-modules/vpc/aws"` from https://registry.terraform.io/browse/modules
  - a git repo `"git::https://example.com/vpc.git?ref=v1.2.0"`

03 Abstractions - Exercise 01

OpenTofu Modules

in `03-abstracations-and-terragrunt/00-modules/`

- in `modules/` there is an `s3-bucket/` module
  - it requires an input variable `"bucket_name"` and

    it provisions an `aws_s3_bucket`, see `main.tf`
  - it also configures the AWS provider in `providers.tf`

Our goal

- we want to use the `s3-bucket` module in `infra/main.tf`

03 Abstractions - Exercise 01

Localstack Reminder

Remember! If you are using **localstack** instead of real AWS:

- in `providers.tf`
  - uncomment the lines below

```
## uncomment when using localstack
```

This applies to all exercises in this chapter!

OpenTofu

LOCALSTACK

03 Abstractions - Exercise 01

OpenTofu Modules

in `03-abstractions-and-terragrunt/00-modules/infra/`

- in `main.tf` add a `module` block to use the `s3-bucket` module:
  within the `module` block…
  - set `source = "path/to/module"` to point to the module (relative path)
  - set the required input: `bucket_name = "<bucket-name>"`
- run `tf init; tf apply`

Find the solution in `01-modules/` for comparison.

3.2 Terragrunt

What's Terragrunt and why use it?

- thin wrapper around OpenTofu / Terraform
  - all `tf` commands can be run with `tg`, e.g. `tf apply` -> `tg apply`
- developed by Gruntwork
  - to simplify infrastructure deployment and
  - plug weaknesses in OpenTofu (Terraform), e.g. the bootstrapping problem
- provides extra levels of abstraction: units and stacks
- simplifies management of complex infrastructure setups, e.g.:
  - multiple regions, multiple environments, multiple accounts

3.2 Terragrunt

⚠️ Note ⚠️

- I have aliased `terragrunt` to `tg` and will use `tg` from here on out
  - `alias tg=terragrunt`
- pay close attention, we will be using both, `tf` and `tg`
  - `tf != tg`

3.2 Terragrunt

Units

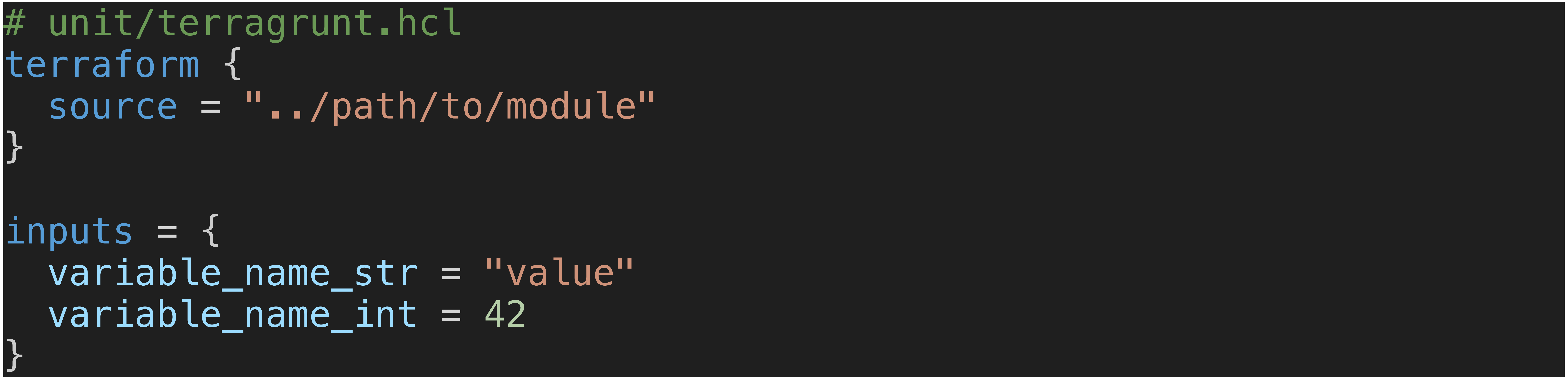
- single instance of infrastructure managed by Terragrunt
- single instance of a module
- has its own state
- a directory containing a `terragrunt.hcl` file is a unit

https://terragrunt.gruntwork.io/docs/getting-started/terminology/#unit

3.2 Terragrunt

Units

- a unit references a single module and sets its inputs



```
# unit/terragrunt.hcl
terraform {
  source = "../path/to/module"
}

inputs = {
  variable_name_str = "value"
  variable_name_int = 42
}
```

- very similar to what we have done in `00-modules/infra/main.tf`

Let’s turn the `infra/` folder into a **Terragrunt unit**…

03 Terragrunt - Exercise 02

Units

in `03-abstractions-and-terragrunt/01-modules/infra/`

- add a `terragrunt.hcl` file and in it…
  - add a `terraform` block with `source` pointing to the `s3-bucket` module
  - add an `inputs` block setting the `bucket_name` variable of the module
- remove the `main.tf` file
- run `tg apply`, type `yes` when prompted and hit ENTER to approve

The `infra/` folder is now a **Terragrunt unit** 🥳

3.3 Terragrunt

Stacks

- collection of units
- typically a stack represents a single application, environment or region
- units in a stack can depend on each other, dictating an update order
- we can update all units in a stack simultaneously with `tg apply -all`

- Side note: Gruntwork is working on replacing the implicit stack based on folder structure with a more explicit `terragrunt.stack.hcl` file
  - https://github.com/gruntwork-io/terragrunt/issues/3313

03 Terragrunt - Exercise 03

Stacks

Our goals

- create a stack with two units
- for now, each unit will create its own S3 bucket
  - …reusing the existing module

03 Terragrunt - Exercise 03

Stacks

in `03-abstractions-and-terragrunt/02-units/`

- add a new folder `stack/`
- move the existing `infra/` folder inside `stack/`: `stack/infra/`
  - optional: rename `infra/` to `unit1/`
- copy `unit1/` to create `stack/unit2/`
  - make sure each unit creates a bucket with a valid and unique name
  - adjust the `source` paths to the module as necessary
- inside `stack/` run `tg apply -all`

03 Terragrunt - Exercise 03

Stacks — Side note

In a real world scenario, we would choose better names! For example…

- `stack/` would likely be named `dev/`, `staging/`, or `prod`
- `unit1/` and `unit2/` would be something like:
  - `frontend-bucket/`
  - `user-data-bucket/`

Terragrunt & Terraform

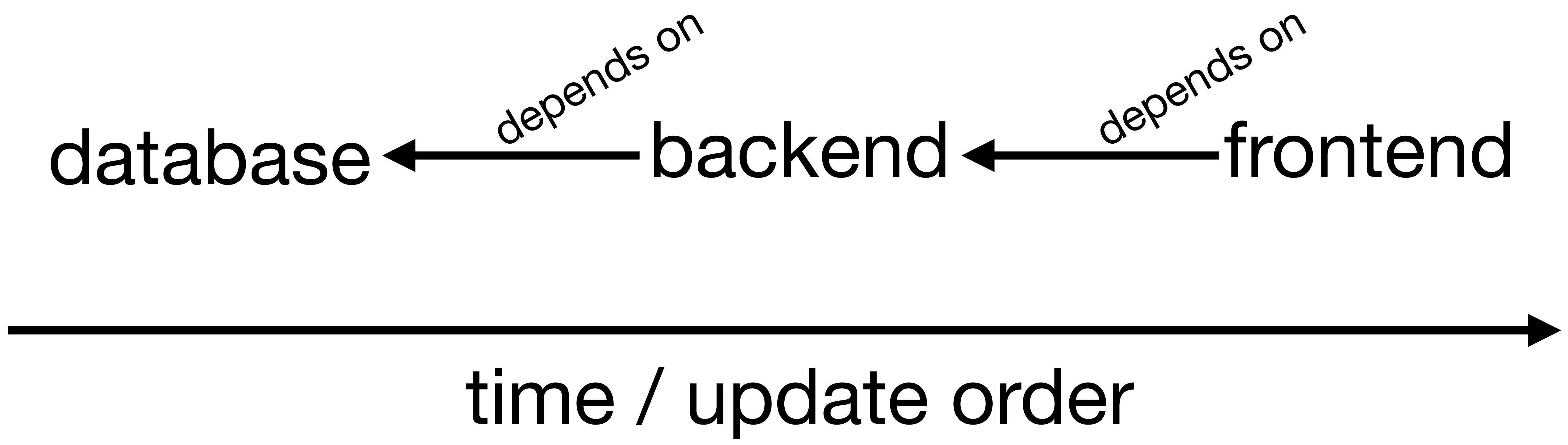
Abstractions Overview

Concept	Framework	How	Use
module	OpenTofu / Terraform	a directory containing `.tf` files	reusable piece(s) of infrastructure
unit	Terragrunt	a directory containing a `terragrunt.hcl` file	instantiates one module and supplies inputs to it
stack	Terragrunt	a directory containing (one or) multiple units (and likely a `root.hcl`)	represents a full application, environment or region

3.4 Terragrunt

Unit dependencies

- when we orchestrate units in a stack, one unit often depends on another
- order to create or update the different pieces of infrastructure is important
- we can dictate the order via `dependency {}` blocks
  - see docs for reference



database ← depends on — backend ← depends on — frontend

time / update order

3.4 Terragrunt

Unit dependencies: example

- consider a backend (unit) that depends on at least a database (unit)

- note that the dependency needs to define outputs

```hcl
# stack/backend-unit/terragrunt.hcl
dependency "db" {
  config_path = "../db" # path to dependency unit
}

inputs = {
  db_host = dependency.db.outputs.host
  db_port = dependency.db.outputs.port
  db_name = dependency.db.outputs.db_name
  db_user = dependency.db.outputs.user
}
```

03 Terragrunt - Exercise 04

Dependencies

Creating two independent buckets is boring. Let’s make this a bit more interesting.

Our goal

- unit1 creates a bucket
- unit2 creates a file in that bucket
  - unit2 depends on unit1

Steps

- create a new module `s3-file/`
- use the new module in unit2

04 Terragrunt - Exercise 04

Dependencies

in `03-abstractions-and-terragrunt/03-stacks/`

- create a second module, e.g. `modules/s3-file/` to create an `aws_s3_object` (think “file”) and add necessary input variables
  - feel free to copy `modules/s3-bucket/` as a baseline
- adapt unit2 to create a file (“object”) in S3 using the new module
- wire the dependencies together (see previous slides for reference)
  - unit2 adds a file in the bucket created by unit1
- run `tg apply -all` in the `stack/` folder

3.5 Terragrunt

Reminder: The bootstrapping problem

requirements

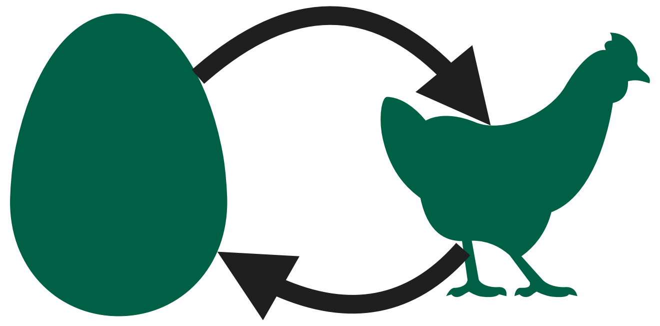
- we want to manage all infrastructure in code
- to enable collaboration we also want to use a remote state

problem

- how do we provision the storage infrastructure for the remote state?

example

- we want to use one S3 bucket solely for the remote state and another S3 bucket for our actual workload



3.5 Terragrunt

Solution to the bootstrapping problem

```
# terragrunt.hcl
remote_state {
  backend = "s3"
  generate = {
    path      = "backend.tf"
    if_exists = "overwrite_terragrunt"
  }

  config = {
    bucket       = "<bucket>"
    region       = "eu-central-1"
    key          = "path/to/tofu.tfstate"
    encrypt      = true
    use_lockfile = true

    ## uncomment when using localstack
    # skip_credentials_validation = true
    # skip_metadata_api_check     = true
    # skip_requesting_account_id  = true
    # endpoints = {
    #   s3 = "http://s3.localhost.localstack.cloud:4566"
    # }
  }
}
```

03 Terragrunt - Exercise 05.1

Bootstrapping

in `03-…/04-dependencies/stack/bucket/`

- in `terragrunt.hcl` add a `remote_state` block to define the state backend (see previous slide)
- localstack: `export AWS_ENDPOINT_URL=http://localhost:4566`
- run `tg apply --backend-bootstrap`
  - when or if prompted, agree to creating the S3 bucket for the remote state
  - ignore `error getting AWS account ID  for bucket …`
    this is a known bug when using terragrunt with localstack: re-run `tg apply`

Where did `terragrunt` put the `backend.tf` file?

03 Terragrunt - Exercise 05.1

Bootstrapping

in `03-…/04-dependencies/stack/bucket/`

- in `terragrunt.hcl` add a `remote_state` block to define the state backend (see previous slide)
- run `tg apply --backend-bootstrap`
  - when or if prompted, agree to creating the S3 bucket for the remote state

Where did `terragrunt` put the `backend.tf` file?

3.5 Terragrunt

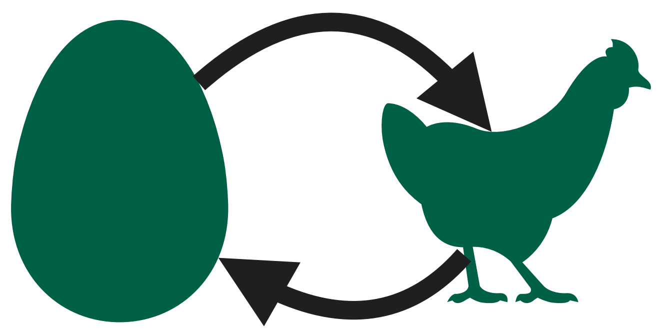
Bootstrapping

Terragrunt …

- creates the state backend, an S3 bucket in our case,
- puts the `backend.tf` file in a `.terragrunt-cache/` subfolder per unit,
- executes OpenTofu commands in the `.terragrunt-cache/` subfolder

Our state backend is created automatically without any hacks 🥳

… but we’re only using remote state in one unit, let’s fix that…



3.5 Terragrunt

Reusable bootstrapping across units in a stack

- let’s make this bootstrapping reusable!
- one of the main reasons for adopting Terragrunt is DRY (don’t repeat yourself)
- an `include {}` block (almost) behaves as if the referenced file was inlined

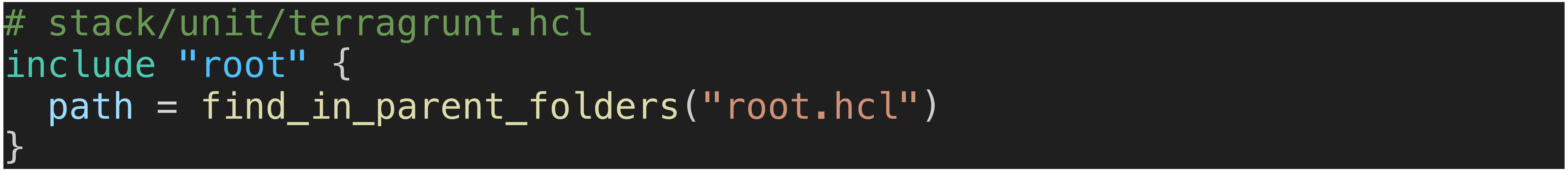
```
# stack/unit/terragrunt.hcl
include "root" {
  path = "../root.hcl”
}
```

read more

3.5 Terragrunt

Reusable bootstrapping across units in a stack

- we can move the `remote_state` block to a reusable `../root.hcl` file
- `find_in_parent_folders()` finds the first `root.hcl` in ancestor folders



```hcl
# stack/unit/terragrunt.hcl
include "root" {
  path = find_in_parent_folders("root.hcl")
}
```

3.5 Terragrunt

Bootstrapping with reusability!

Our goals

- one bucket for all states of one stack
- one state file for each unit

Steps

- move `remote_state` block to `stack/root.hcl` (and include it)
- create a dynamic state file key by utilizing a built-in terragrunt function `path_relative_to_include()`
  - it returns a unit's directory name

03 Terragrunt - Exercise 05.2

Bootstrapping

in `03-…/05-01-bootstrapping/stack/`

- create a `root.hcl` file
- cut the `remote_state` block from `bucket/terragrunt.hcl` and paste it in `root.hcl`
  - utilize `path_relative_to_include()` in the state `key`
- in both units, `bucket/` and `file/`, …
  - add and configure an `include "root" {}` block in `terragrunt.hcl`
  - run `tg apply` in both units

3.6 Terragrunt

Generating the provider

- both modules, `s3-bucket/` and `s3-file/`, configure a `provider`
- we want to keep our code DRY (don’t repeat yourself)
- terragrunt let’s us generate files with arbitrary contents with a `generate {}` block

```
generate "provider" {
  path = "provider.tf"
  if_exists = "overwrite_terragrunt"
  contents = <<EOF
FILE CONTENTS
EOF
}
```

BONUS

03 Terragrunt - Exercise 06

Generating the provider — Bonus

in `03-…/05-02-bootstrapping`

- in `stack/root.hcl` add a `generate "provider" {}` block
  - take a look at the modules’ `provider.tf` files for reference
  - don’t forget the part specifying the provider version
- remove all `provider.tf` files in `modules/`
- run `tg apply -all` in `stack/`

3 Terragrunt

Wrap up

- Terragrunt is a powerful wrapper to enable DRY infra code
- it handles some of OpenTofu's shortcomings like the bootstrapping problem
- it provides additional abstractions to simplify management of complex infrastructure setups
- stacks > units > modules
  - low level resources are defined in **modules**
  - **modules** are instantiated and configured in **units**
  - **units** are orchestrated in **stacks**

3 Terragrunt

Reminder: Save costs!

- run `tg destroy -all`
- verify with `aws s3 ls` that all your buckets are deleted
- delete buckets for storing state
    - use script: `scripts/empty-and-delete-s3-bucket.sh <bucket>` or
    - via the AWS console

3 Terragrunt

Criticism and Alternatives

Is Terragrunt overkill? Do I need it?

- Redit: Should I use Terragrunt?
- Medium: Why use Terragrunt if you already use Terraform Modules?
- Alternatives:
  - OpenTofu Workspaces: State / Workspaces, CLI / Workspaces
    - Issue / Discussion to abolish OpenTofu workspaces
  - https://terramate.io/
  - https://terraspace.cloud/

3 Terragrunt

Resources

- Tutorials: Quick Start, Overview, Terralith to Terragrunt
- Terminology: Module, Unit, Stack
- Modules registry: AWS Modules
- Features: Units, Stacks, includes
- HCL: Dependencies, Generate, Functions

Agenda

- Introductions
- Setup prerequisites

1. OpenTofu Basics
2. State management
3. Abstractions and Terragrunt
4. **Deploying a simple web app**
5. Final Exercise

4. Deploying a simple web app

4. Deploying a Simple Web App

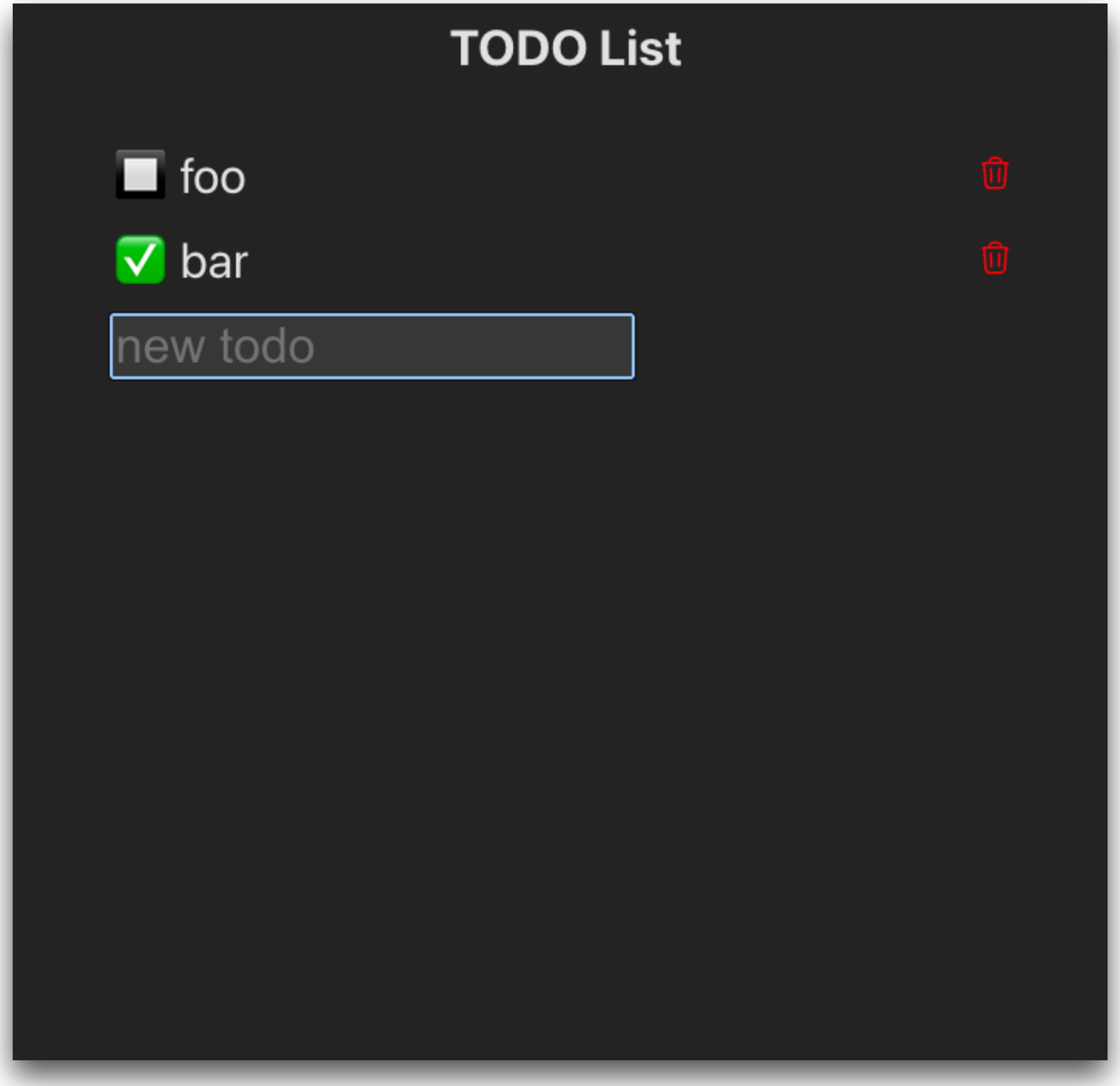
Agenda

- deploy a simple Todo list web app
  - using AWS in `03-deploy-web-app/01-aws` or
  - using docker locally in `03-deploy-web-app/01-docker`
    - since we need a runtime / compute, localstack won’t work
      (at least not with the free version)
- at the end of this chapter you should have a running and accessible Todo list app with persisted data

4. Deploying a Simple Web App

Todo list app

- in the app you can
  - add todo items
  - check them off
  - and delete them
- all data is persisted in a database
- … I may eventually make it prettier 😄
  for now it's functional enough



TODO List

⬜ foo

✅ bar

4. Deploying a Simple Web App

- the app may run…
  - locally with Docker and only be accessible locally or
  - in the public AWS cloud and be globally accessible
    - if you own a domain and / or can set DNS records, you can even have it running at a human readable (sub-)domain with HTTPS
- this chapter is a precursor for the last chapter / final exercise
  - there is not that much to do once the prerequisites are setup correctly
  - continue with DOCKER or AWS

4. Deploying a Simple Web App

Prerequisites for Docker

- docker needs to be installed
  - https://www.docker.com/get-started/
- docker needs to be running
- you need to know the path to the docker socket on your machine
  - e.g.: `unix:///var/run/docker.sock`
  - on MacOS: `unix:///Users/USERNAME/.docker/run/docker.sock`

04 Simple Web App - Exercise 01

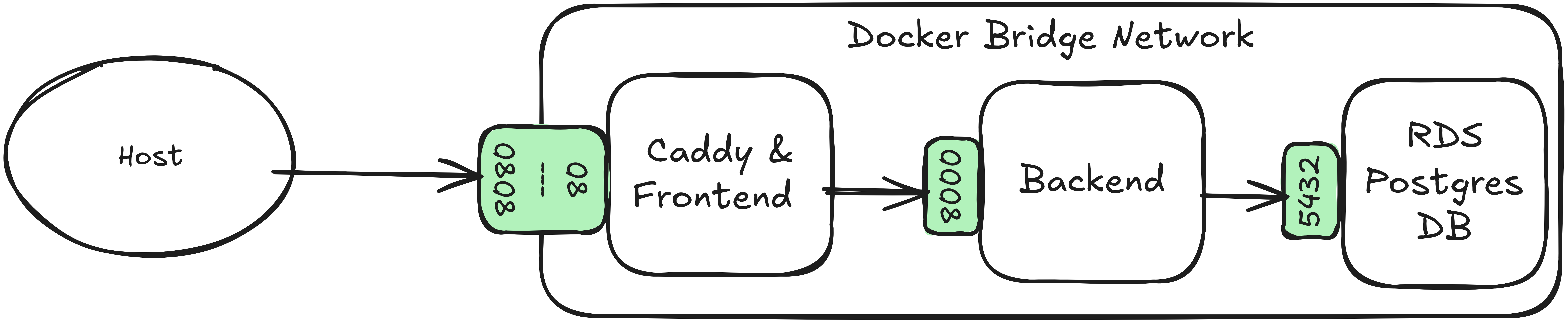
with OpenTofu and Docker

- navigate to `03-deploy-web-app/01-docker`
- `cp .env.example .env`
- in `.env` set a database password and the path to the Docker socket
- execute `source .env; tf init; tf apply`
- open `localhost:8080` in a browser (it may take a few minutes)
  - execute `docker exec -it todo-list-app-frontend top` and wait for `caddy` to show up in the `COMMAND` column,
    then refresh the browser page

4. Deploying a Simple Web App

with OpenTofu and Docker

- what did we deploy?



4. Deploying a Simple Web App

with OpenTofu and Docker

Details and new concepts

- `provisioner "local-exec"` to execute command locally
- uploading single files to containers and mounting volumes in them
- setting environment variables
- running init scripts to
  - install packages and dependencies necessary to run our software
  - and start our software

4. Deploying a Simple Web App

Disclaimer!

- the architecture / setup was intentionally kept simple
- improvements
  - build own docker images for frontend and backend

4. Deploying a Simple Web App

Prerequisites for AWS: new ssh key

- generate a new ssh key pair with `ssh-keygen`
- `ssh-keygen -t ed25519 -a 420 -C "name@example.com"`
  - file location and name, e.g `~/.ssh/aws-iac-workshop.ed25519`
  - 🚨 DO NOT set a password 🚨 for simplicity
    - for a best practice setup read these slides

04 Simple Web App - Exercise 01.1

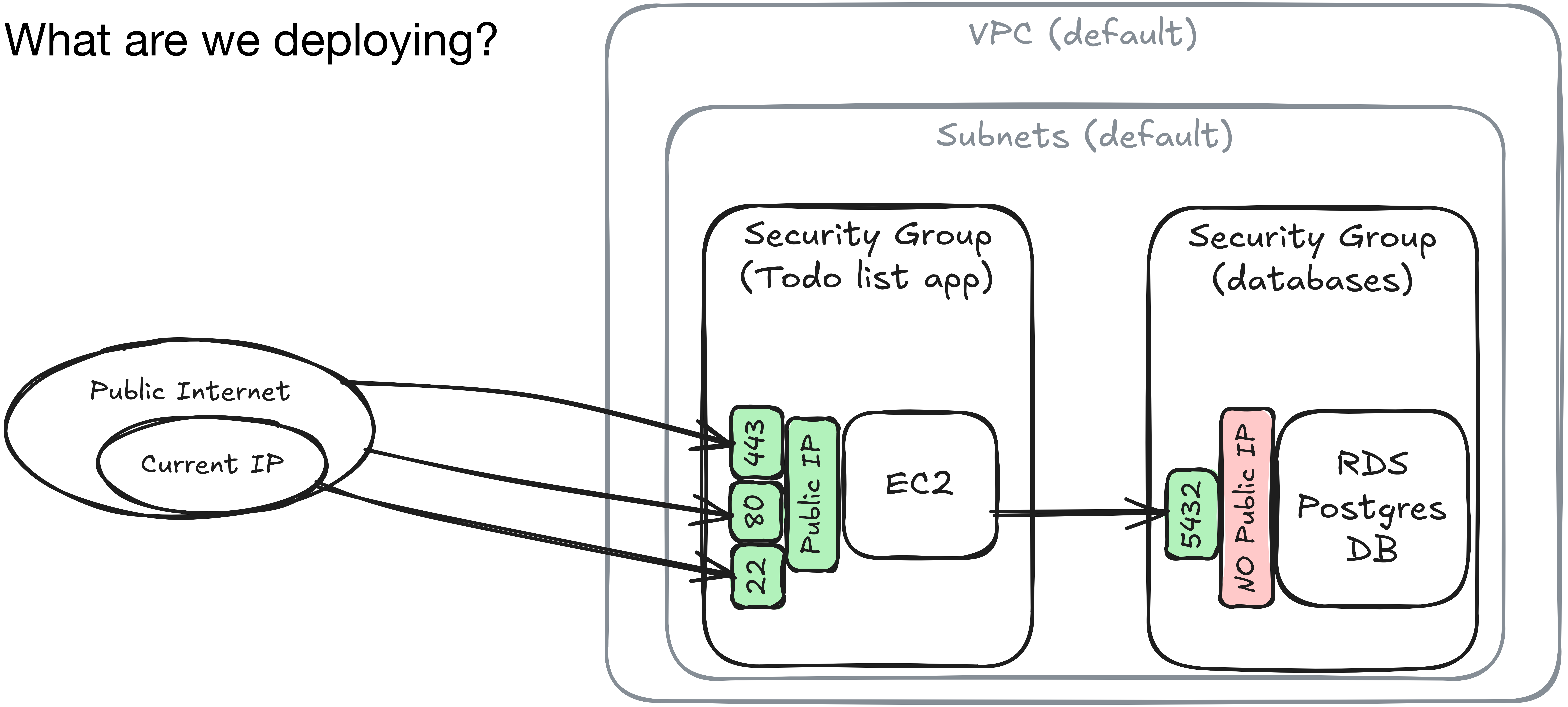
with OpenTofu and AWS

- navigate to `03-deploy-web-app/01-aws`
- `cp .env.example .env`
- in `.env` set a database password and the paths your ssh-key (both public and private), e.g. `~/.ssh/aws-iac-workshop.ed25519(.pub)`
- execute `source .env; tf init; tf apply` (it'll take a while)
  - once completed: take note of the outputs, enter the `ec2_domain` in a browser (use `http://` **not** `https://` for now)
  - as we wait, let's see what we're deploying on the next slide

4. Deploying a Simple Web App

with OpenTofu and AWS

- What are we deploying?



AWS

4. Deploying a Simple Web App

with OpenTofu and AWS

Details and new concepts

- `provisioner "local-exec"` to execute command locally
- `provisioner "remote-exec"` to execute command on remote machine
- `provisioner "file"` to copy files and directories onto remote machine
- requires `connection { type = "ssh"}`

04 Simple Web App - Exercise 01.2

Bonus 🌟: HTTPS with access to DNS records

If you have access to a domain and / or DNS records

- add an `A` record for `todo.your-domain.eu` pointing to `<ec2_ip>`
- open `http://todo.your-domain.eu` in a browser

Let’s enable HTTP**S**

- log into the ec2 machine: run
  `ssh ubuntu@<ec2_domain> -i ~/.ssh/aws-iac-workshop.ed25519`
- edit `/etc/caddy/Caddyfile`
  - replace `:80` with `todo.your-domain.eu`
  - restart the caddy service: `sudo systemctl restart caddy`
  - open `https://todo.your-domain.eu` in a browser

4. Deploying a Simple Web App

Disclaimer!

- the architecture was intentionally kept simple
- improvement suggestions and options (some are mutually exclusive)
  - separate public and private subnets
  - separate frontend and backend (machines) (and reverse proxy)
  - serve static frontend files from S3 via CloudFront
  - add autoscaling w/ or w/o a load lalancer, potentially using a service other than EC2, e.g. Lambda, Elastic Beanstalk
  - use custom Docker images and a service that allows running them
  - use Ansible to configure EC2 machines once they're spun up via OpenTofu

04 Simple Web App - Exercise 01.3

destroy to save costs

- run `tf destroy`
- in chapter 5 we will come back to this and deploy this in two different environments

4. Deploying a simple web app

Wrap up

- spin up multiple pieces of infrastructure with just **one command** 😁🥳 (once everything is setup)
- this is very easily reproducible and reusable (see chapters 4 and 5)
- I hope you find this kind of automation as exciting and useful as I do! 😁

4. Deploying a simple web app

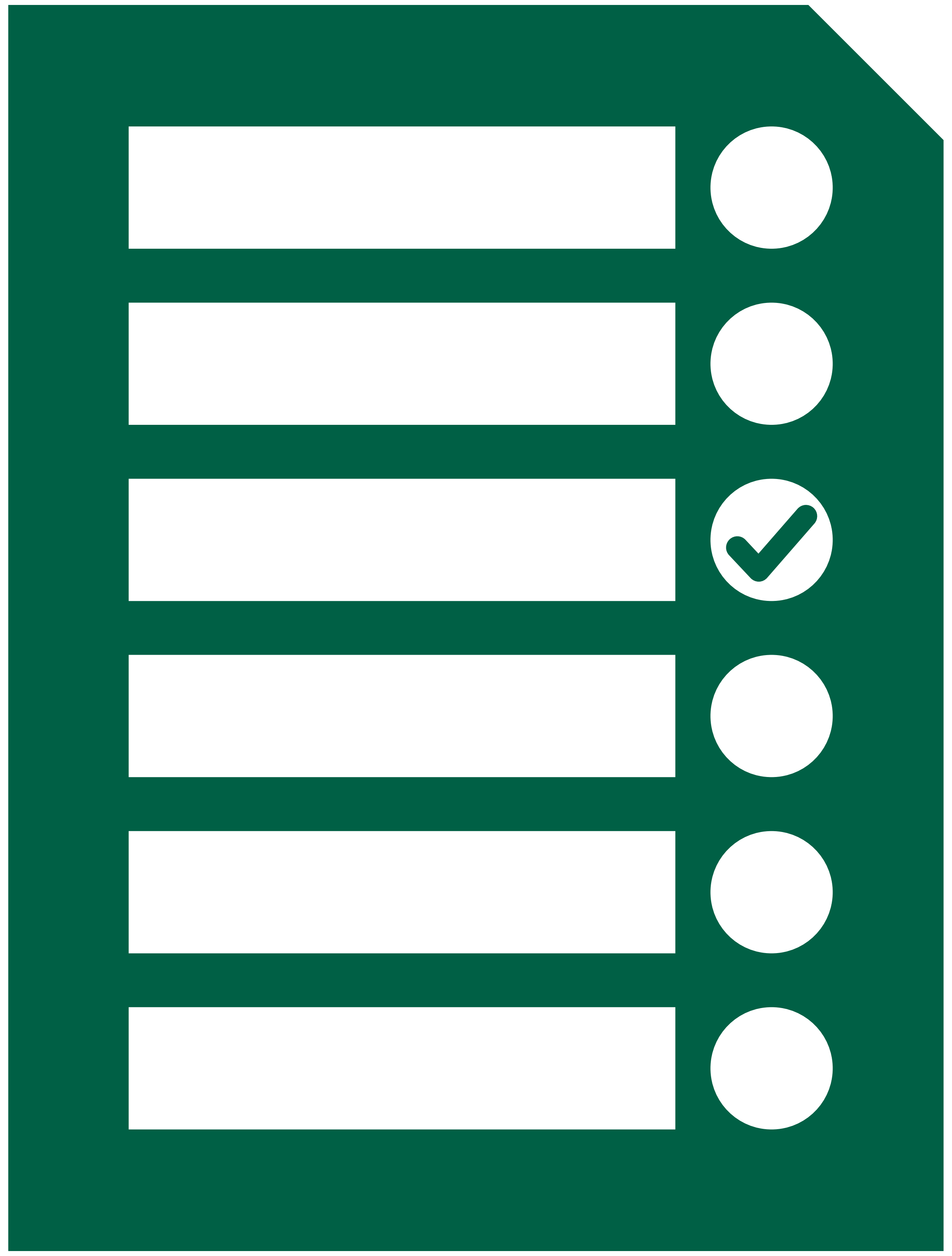
Resources

- Provisioners (as a last resort)
    - File Provisioner
    - local-exec Provisioner
    - remote-exec Provisioner
- Best Practice SSH setup

Agenda

- Introductions
- Setup prerequisites

1. OpenTofu Basics
2. State management
3. Abstractions and Terragrunt
4. Deploying a simple web app
5. **Final Exercise**



5. Final Exercise

5. Final Exercise

Let’s put it to the test

- you (hopefully) have learned everything you need to finish the final exercise all by yourselves:
  - resources, providers, and modules
  - remote state
  - applying infrastructure changes
  - Terragrunt abstractions
- you will deploy the web app from chapter 3 all by yourselves

5. Final Exercise

Goals

- Deploy the web app from chapter 3 in **two different environments***

How

- split the web app into modules and units and define dependencies
- use stacks to orchestrate and deploy the app in different environments*

Optional shortcuts (if short on time):

- use larger modules / units or even just a single one for the entire app
- use local state rather than remote state

*best practice: use one AWS account per environment, alternatively use different **regions** to emulate it

5. Final Exercise

Goals

- Deploy the web app from chapter 3 in **two different environments***

How

- split the web app into modules and units and define dependencies
- use stacks to orchestrate and deploy the app in different environments*

Optional shortcuts (if short on time):

- use larger modules / units or even just a single one for the entire app
- use local state rather than remote state

*to emulate different environments use separate docker networks and different container names, e.g. “container-${var.env}”

Terragrunt & Terraform

Abstractions Overview

Concept	Framework	How	Use
module	OpenTofu / Terraform	a directory containing `.tf` files	reusable piece(s) of infrastructure
unit	Terragrunt	a directory containing a `terragrunt.hcl` file	instantiates one module and supplies inputs to it
stack	Terragrunt	a directory containing (one or) multiple units (and likely a `root.hcl`)	represents a full application, environment or region

5. Final Exercise

Hints and solutions

- the following slides contain hints
- feel free to take a peek if you’re stuck or ask for help or hints
- solutions for both AWS and Docker versions are also in the repo
  - `05-final-exercise-(aws|docker)-solution`

5. Final Exercise

Hint 1: Folder structure (suggestion)

- `05-final-exercise/`
  - `stacks/`
    - `dev/`
      - `root.hcl`
      - `<unit>/terragrunt.hcl`
    - `prod/`
      - `root.hcl`
      - `<unit>/terragrunt.hcl`
  - `modules/`
    - `networking/*.tf`
    - `db/*.tf`
    - `server/*.tf`

5. Final Exercise

Hint 1: Folder structure (suggestion)

- `05-final-exercise/`
  - `stacks/`
    - `dev/`
      - `root.hcl`
      - `<unit>/terragrunt.hcl`
    - `prod/`
      - `root.hcl`
      - `<unit>/terragrunt.hcl`
  - `modules/`
    - `networking/*.tf`
    - `db/*.tf`
    - `backend/*.tf`
    - `frontend/*.tf`

5. Final Exercise

Hint 2: Terragrunt functions and features

- the following Terragrunt functions and features may be useful to make thing dynamic
  - `extra_arguments` with `get_terraform_commands_that_need_vars()`
  - use `locals` in `root.hcl` and `terragrunt.hcl` files
  - expose `locals` from `include`d file(s) with `expose=true`

5. Final Exercise

Notes on best practices

- deploy different environments in different accounts of your cloud provider
    - isolate resources from dev, staging and prod environments
    - use AWS Organizations to centralize account management and billing
- avoid using AWS ACCESS KEY and SECRET KEY, use temporary credentials
    - use AWS IAM Identity Center for SSO and access to your accounts

Thank you for your participation!

I hope you have learned
some things — or rather a lot!

Feedback

Feedback

I would love to hear your feedback!

- What did you learn?
- What was your favorite part?
- **What can I improve?**

Let’s connect!
I’d appreciate a shoutout on LinkedIn :)

If you prefer, feel free to send me your feedback privately: patrick@patrick-moelk.eu

Patrick Mölk

Code Smart. Test Hard. Deploy Fast. Build Infrastructure That Lasts.

Cloud Infrastructure, DevOps, CI/CD, Automation, Backend

Freelance IT Consultant and Software Developer

Seven years of professional software development experience. **Let's connect!**

https://patrick-moelk.eu/

https://linkedin.com/in/patrick-moelk/

https://mastodon.social/@patrick_moelk

More useful concepts

Not covered, but useful

- `count`, `for_each`: useful to create dynamic number of resources
- `tf plan`: useful for CI/CD contexts, see what will change before running pipeline
  - e.g. in GitLab merge requests before merging (GitLab iac integration)
- `tf taint`: marks resource as “tainted”, forces resource replacement
- `tf apply -auto-approve` / `tf apply <plan-file>`: useful in CI/CD
- `tf console`: interactive console, useful to test tf code / evaluations
- A comprehensive guide to managing secrets in your Terraform code

“Deleted Scenes”

Slides that were removed or simplified due to time constraints

0.2 AWS Credentials (SSO)

Setup Prerequisites

Best practice: Side note on AWS credentials

- consider “access key id” and “secret access key” to be deprecated / unsafe
- use role-based temporary credentials instead,
  - e.g. with AWS Identity Center (formerly AWS SSO)
- we use “access key id” and “secret access key” here for simplicity
  - especially when working with localstack rather than the real AWS
  - if you have SSO setup in your AWS account(s), use it

Setup Prerequisites

AWS profile for LOCALSTACK

If you do **not** have a real AWS account available

in `~/.aws/config` (if not present create the directory and file)

- copy and paste the following *dummy* credentials

```
[profile opentofu-workshop]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = 1234567890123456789012345678901234567890
aws_account_id = 123456789012
region = eu-central-1
```

Setup Prerequisites

AWS profile: without SSO

If you DO NOT have SSO setup

in `~/.aws/config`

- generate `AWS_ACCESS_KEY_ID` and `AWS_SECRET_KEY` in the AWS console

```
[profile opentofu-workshop]
aws_access_key_id = <AWS_ACCESS_KEY_ID>
aws_secret_access_key = <AWS_SECRET_KEY>
aws_account_id = <aws-account-id>
region = eu-central-1
```

Setup Prerequisites

AWS profile: with SSO

If you DO have SSO setup

- if not already done, run
  `aws configure sso --profile opentofu-workshop`
  and follow the prompts

your `~/.aws/config` should contain something like this

```
[profile opentofu-workshop]
sso_start_url = https://d-xxx.awsapps.com/start/#
sso_region = eu-central-1
sso_account_id = <account-id>
sso_role_name = <some-role-name>
```

Setup Prerequisites

AWS profile: with SSO

- run `aws sso login --profile opentofu-workshop` to generate temporary credentials
- set `AWS_PROFILE=opentofu-workshop` env variable to use the temporary credentials
- run `aws sso logout` to log out of ALL profiles’ sessions

1.2 AWS Credentials (Exercise)

01 Basics - Exercise 02

Secrets

- We still need to find a solution to keep secrets out of our code base
  - especially out of version controlled files
- reminders
  - we can set the `sensitive` flag on variables to prevent printing secrets
  - we can set variables by setting environment variables like `TF_VAR_<var-name>=secret`

01 Basics - Exercise 02.4

local variables and expressions

in `01-basics/01-providers-and-resources`

- find a convenient solution to keep secrets outside of version control

- hints:
  - utilize `TF_VAR_<variable-name>=secret` environment variables
  - take a look at the root `.gitignore`

Solution is on next slide

01 Basics - Exercise 02.4 cont'd

local variables and expressions

in `01-basics/01-providers-and-resources`

- find a convenient solution to keep secrets outside of version control:
  - create a `.env` file which is not tracked by git (see `.gitignore`)
  - set secrets in the `.env` file
    - `export TF_VAR_aws_access_key_id=YOUR-ACCESS-KEY-ID`
    - `export TF_VAR_aws_secret_key=YOUR-SECRET-KEY`
  - add more secrets to the `.env` file if necessary
  - run `source .env` before `tf apply`

1.2+ Data sources

1.3 Data sources

Cleanup

- Data sources can be used to get data or information
- Data sources do not create resources
- useful for querying or fetching data and supplying it to resources as inputs
- example: use your user name to create a dynamic S3 bucket name
  - using the `whoami` command

```
data "local_command" "name" {
  command = "whoami"
}
```

01 Basics - Exercise 02.6

Bonus

in `01-basics/02-variables-and-outputs`

- `s3.tf` add a `data "local_command" {}` block to get your username
  - `command = "whoami"`
- adapt the `var.s3_bucket_name` and the `local.bucket_name` definition
- run `tf apply`
  - address any errors and try again
    - you’ll likely need to run `tf init -upgrade`

4.6 Terragrunt dependencies

3.4 Terragrunt

Unit dependencies

- when we orchestrate units in a stack, one unit often depends on another
- terragrunt needs to know in which order to create the pieces of infrastructure
- we can do this via a `dependency {}` block
  - see docs for reference
- for less explicit dependencies, when one unit does **not** depend on the outputs of the another unit there is a `dependencies {}` block
  - see docs for reference

4 Deploying a simple web app (Best practice SSH Setup)

4. Deploying a Simple Web App

Prerequisites for AWS

Steps

- generate a new ssh key pair with `ssh-keygen`
- add identity to ssh-agent with `ssh-add`
- configure AWS host in `~/.ssh/config`

4. Deploying a Simple Web App

Prerequisites for AWS: new ssh key

generate a new ssh key pair with `ssh-keygen`

- `ssh-keygen -t ed25519 -a 420 -C "name@example.com (IaC-workshop-AWS)"`
- `~/.ssh/aws-iac-workshop.ed25519`
- set a password!

4. Deploying a Simple Web App

Prerequisites for AWS: add key to ssh-agent

- `ssh-add ~/.ssh/aws-iac-workshop.ed25519`
  - enter the password
- Mac users: make use of Keychain with the `--apple-use-keychain` flag
  - enter the password and it will be added to your Keychain
  - remove all identities: `ssh-add -D`
  - run `ssh-add ~/.ssh/aws-iac-workshop.ed25519 --apple-use-keychain` again
    - you won't be prompted for a password again

4. Deploying a Simple Web App

Prerequisites for AWS: ~/.ssh/config

- add the following lines in `~/.ssh/config`

```
Host ec2-*.compute.amazonaws.com
  User ubuntu
  UseKeychain yes # <-- MacOS
  PreferredAuthentications publickey
  IdentityFile ~/.ssh/aws-iac-workshop.ed25519
```

- this will allow you to enter `ssh ec2-xxx.compute.amazonaws.com` without specifying a user or ssh key
  - and on MacOS without being prompted for a password
- https://media.ccc.de/v/gpn21-28-noch-besser-leben-mit-ssh#t=314 (German)

4. Deploying a Simple Web App

SSH: Secure and convenient

If you’re interested to learn more about SSH setups and workflows

- Leyrer has given some great talks on how to setup ssh in a safe and convenient way (talks are in German)
  - https://media.ccc.de/v/gpn20-8-besser-leben-mit-ssh
    - https://martin.leyrer.priv.at/downloads/talks/2022/2022-05%20-%20gpn20%20-%20Besser%20leben%20mit%20SSH.pdf
  - https://media.ccc.de/v/gpn21-28-noch-besser-leben-mit-ssh#t=314