- acend gmbh

Setup

The Terraform lab exercises require the installation and configuration of a few applications. This page will help you getting started.

Web shell

Your Terraform lab trainer will provide you personal credentials to your web shell.
All required CLI tools and an IDE are installed and ready to use.

Local installation

The exercises assume a UNIX environment. In case you are working under Windows, be advised to install the **Windows Subsystem for Linux** as documented here: https://docs.microsoft.com/en-us/windows/wsl/install-win10

CLI Tools

Please install the following applications:

- `terraform` - Terraform CLI
  There are two methods for installing `terraform`:
  - **Recommended:** Install and manage different versions with `tfenv` from https://github.com/tfutils/tfenv
    Run the following commands to install the latest version of Terraform:

    ```
    tfenv install latest
    tfenv use latest
    ```

  - **Alternative:** Follow the instructions on the Hashicorp website at https://learn.hashicorp.com/tutorials/terraform/install-cli
- `az` - Azure CLI
  Note: This is used for the Azure workshop only!
  See https://docs.microsoft.com/en-us/cli/azure/install-azure-cli
- `kubectl` - Kubernetes CLI
  See https://kubernetes.io/docs/tasks/tools/
- `helm` - Helm - Kubernetes package manager CLI (optional) See https://helm.sh/docs/intro/install/
- `jq` - JSON query CLI (optional)
  See https://stedolan.github.io/jq/download/

Make sure `terraform` is installed correctly and found in your `PATH` by running:

```
terraform version
```

Optional: To install `bash` autocompletion, run the following command and restart your shell:

```
echo "complete -C `which terraform` terraform" >> ~/.bashrc
```

IDE

Install a text editor of your choice. PyCharm Community Edition IDE with the HCL plugin is recommended for its powerful features like resource and attribute auto-complete, refactoring etc.

PyCharm

To install PyCharm, follow the instructions:

- Goto https://www.jetbrains.com/pycharm/download
- add plugin **HashiCorp Terraform / HCL language support**

Visual Studio Code

Visual Studio Code offers Terraform support via extension, follow the instructions:

- Goto https://code.visualstudio.com/download
- add the extension **HashiCorp Terraform**

Labs

In this training, you’re going to learn the basics behind Terraform technology.

- Introductory presentation: Terraform Indroduction?
- Install Terraform on your computer
- Learn Terraform Basics
- Terraform advanced topics
- Use Terraform in the Cloud
- Terraform Cloud examples

- acend gmbh

1. Introduction

Welcome to the Terraform training lab!

What is Terraform?

Terraform is an open-source infrastructure-as-code software tool created by HashiCorp, that provides a consistent CLI workflow to manage hundreds of cloud services. Terraform codifies cloud APIs into declarative configuration files.

Useful Links

- Terraform Docs
- Terraform Registry & Modules
- Terraform Tutorials

Terraform Infrastructure-as-Code (IaC)

Terraform code is written in HCL (HashiCorp Configuration Language) which is technically not source “code” but configuration. The definition of all resources for your infrastructure is defined in `.tf` files in the same directory. Sub-directories are used to store parameters or Terraform modules, but we’ll come to that later.

The filename does not serve special purpose; Terraform internally merges all files ending with `.tf`. Choose filenames which are expressive and meaningful for other engineers to navigate your code.

A typical project structure looks as followed:

- `main.tf`
- `variables.tf`
- `outputs.tf`
- `versions.tf`
- `[component].tf`

In the next lab chapters you will create these files and understand what to place in these files.

2. First steps

> **Important**
> Please make sure you completed the setup before you continue with this lab.

First Steps

Start your IDE in an empty project directory and launch a UNIX shell.

The upcoming labs will always refer to the root folder of your exercises. Store it in an environment variable to access it quicker:

```
export LAB_ROOT=`pwd`
```

Now create a new directory:

```
mkdir $LAB_ROOT/first_steps
cd $LAB_ROOT/first_steps
```

Create a new file named `main.tf` in your working directory and paste the following:

```
output "hello" {
  value = "Hello Terraform!"
}
```

Now run the commands

```
terraform init
terraform apply
```

Terraform asks for your confirmation, enter `yes`:

```
...
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

Well done! You created your first “Hello World!” in Terraform.
The next chapters will explain what we’ve actually just done here - let’s move on!

- acend gmbh

3. Basics

We will learn the basic syntax of Terraform HCL and use commands to initialize, create and destroy resources.

3.1. Resources

Preparation

Create a new directory for this exercise:

```
mkdir -p $LAB_ROOT/basics/resources
cd $LAB_ROOT/basics/resources
```

Step 3.1.1: Create `main.tf`

We will start with a simple example by creating a resource of type `random_integer`. This resources generates a random number in the configured range.

Create a new file named `main.tf` in your working directory and paste the following:

```
resource "random_integer" "number" {
  min = 1000
  max = 9999
}
```

Explanation

The `resource` block defines one (or multiple) infrastructure objects which are managed by Terraform.

For more information about Terraform resources, please see
https://www.terraform.io/docs/language/resources/syntax.html

Step 3.1.2: Init Terraform

Download all required Terraform providers and initialize the local state:

```
terraform init
```

Output:

```
Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/random...
- Installing hashicorp/random v3.5.1...
- Installed hashicorp/random v3.5.1 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!
```

Step 3.1.3: Plan execution

The planing will help Terraform to understand your configuration and verify the syntax. To create a provisioning plan, run:

```
terraform plan
```

This will show output similar to:

```
Terraform will perform the following actions:

  # random_integer.acr will be created
  + resource "random_integer" "number" {
      + id     = (known after apply)
      + min    = 1000
      + max    = 9999
      + result = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

Step 3.1.4: Apply configuration

After planing the infrastructure provisioning, we instruct Terraform to apply the configuration:

```
terraform apply
```

Terraform will print the execution plan again and ask for confirmation. Type `yes` to continue.

```
random_integer.number: Creating...
random_integer.number: Creation complete after 0s [id=9437]

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

Step 3.1.5: Inspect the local state

After creating the resources you might wonder, where Terraform stores the generated number? As we are not in the cloud yet, where is the state stored?

Run the following command:

```
ls -l
```

There is a file called `terraform.tfstate` which contains all information of your resources provisioned by Terraform. Your random number is stored in this file. Terraform requires a `.tfstate` file to store all your configurations. It is used to compare your desired state (in code) against the real world (fetched by APIs) and last execution (stored in the state) plus objects not available by API resource like random passwords, SSL certs (also stored in the state).

In a later chapter we will learn how store this file in the cloud and why it is best practice.

Step 3.1.6: Destruction

To remove or de-provision all resources, run the following command:

```
terraform destroy
```

Terraform will again ask for confirmation if you want destroy the content. Type `yes` to destroy all resources managed by this Terraform code base (aka. stack).

3.2. Variables

Preparation

Create a new directory for this exercise:

```
mkdir -p $LAB_ROOT/basics/variables
cd $LAB_ROOT/basics/variables
```

Step 3.2.1: Create variables.tf and main.tf

Create a new file named `variables.tf` in your working directory and add the following content:

```
variable "random_min_value" {
  type        = number
  default     = 1000
  description = "min value of the random number"
}
```

Create a new file named `main.tf` in your working directory and add the following content:

```
resource "random_integer" "number" {
  min = var.random_min_value
  max = 9999
}
```

Explanation

It is best practice putting all required input variables in the file `variables.tf` .

The `type` and `description` arguments are optional but good practice; don't overdo the `description` tho, nobody really reads it...

Step 3.2.2: Apply the configuration

Run the commands

```
terraform init
terraform apply
```

Step 3.2.3: Change the default value

To see how Terraform applies changes to your existing resources, change the `default` value of

`random_min_value` to `2000` in the `variables.tf` file:

```
variable "random_min_value" {
  type        = number
  default     = 2000
  description = "min value of the random number"
}
```

Then run the command

```
terraform apply
```

And terraform will display the required changes to create the state in your code. You will see a similar plan like this:

```
random_integer.number: Refreshing state... [id=8731]

Terraform used the selected providers to generate the following
execution plan. Resource actions are indicated with the following
symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # random_integer.number must be replaced
-/+ resource "random_integer" "number" {
      ~ id     = "8731" -> (known after apply)
      ~ min    = 1000 -> 2000 # forces replacement
      ~ result = 8731 -> (known after apply)
        # (1 unchanged attribute hidden)
    }

Plan: 1 to add, 0 to change, 1 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

Step 3.2.4: Add a local variable

Sometimes you want to modify or derive a value from a variable. This can be achieved by declaring a “local” variable in a `locals` block. Add the following on the first line of `variables.tf` :

```
locals {
  random_max_value = var.random_min_value + 31337
}
```

Then modify the `resource` block in `main.tf` as followed:

```
resource "random_integer" "number" {
  min = var.random_min_value
  max = local.random_max_value
}
```

Try it out

Remove the `default = 2000` statement from the block and run `terraform apply` .

3.3. Outputs

Preparation

Finish the Variables exercise and navigate to the directory:

```
cd $LAB_ROOT/basics/variables
```

Step 3.3.1: Create outputs.tf

Create a new file named `outputs.tf` in your working directory and add the following content:

```
output "number" {
  value = random_integer.number.result
  description = "random value created by terraform"
}
```

Step 3.3.2: Apply the configuration

Run the command

```
terraform apply
```

and you should see output similar to this:

```
Plan: 0 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + number = 15670

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes


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

Outputs:

number = 15670
```

Step 3.3.3: Access the output

If you just want to access the output value without running apply, you can just run:

```
terraform output number
terraform output -raw number
```

Can you spot the difference between the outputs?

Step 3.3.4: Handling sensitive output

Add the `sensitive` keyword to the `outputs.tf` file as followed:

```
output "number" {
  value       = "The number is ${random_integer.number.result}"
  description = "random value created by terraform"
  sensitive   = true
}
```

This will mask the console output of the value. The output is still available by explicitly specifying the name as followed:

```
terraform output number
```

Try it out

You can also print the the output in json format and use tools like `jq` to process it further:

```
terraform output -json | jq '.number.value'
```

This is useful when handling large JSON data structures.

> **Note**
> `terraform output` can be used to create input or configuration for other CLI tools like Ansible.

3.4. Data Sources

Preparation

Create a new directory for this exercise:

```
mkdir -p $LAB_ROOT/basics/data_sources
cd $LAB_ROOT/basics/data_sources
```

Step 3.4.1: Create resources

Create a new file named `main.tf` in your working directory and paste the following:

```
resource "random_integer" "number" {
  min = 1000
  max = 9999
}

resource "local_file" "random" {
    content  = random_integer.number.result
    filename = "random.txt"
}
```

Step 3.4.2: Apply the configuration

Run the commands

```
terraform init
terraform apply
```

You will see on the console the resource `random_integer.number` is created **before** the `local_file.random` because the `result` attribute of the random integer is passed as content.

This shows the dependency tracking and resolution of Terraform in action.

Step 3.4.3: Taint a resource

Sometimes you want to recreate a specific resource. Terraform offers the `taint` command to mark a resource for recreation and `untaint` to remove the mark.

> **Note**
> The `taint` command is rarely used in practice.

Important: The next apply will destroy and create the resource which might lead to a recreation of other depending resources!

```
terraform taint random_integer.number
```

Since Terraform 0.15.2 you also can do this with the option `-replace <terraform object name>` :

```
terraform apply -replace="random_integer.number"
```

The random number should now be recreated.

Step 3.4.4: Reference an existing resource

Create a new file in your current working directory:

```
echo terraform4ever > propaganda.txt
```

Now add the following code to `main.tf` :

```
data "local_file" "propaganda" {
  filename = "propaganda.txt"
}
```

Create a new file `outputs.tf` and add the following content:

```
output "propaganda" {
  value = data.local_file.propaganda.content_base64
}
```

Run the command:

```
terraform apply
```

And you should see the base64 encoded version of our referenced file `propaganda.txt`

Explanation

The `data` keyword references objects not managed by this terraform stack (code base). This is common and very useful in cloud engineering to reference already existing infrastructure components like manually added DNS zones or resources managed by another Terraform stack!

Try it out

- acend gmbh

You can run the following command to base64 decode the output:

```
terraform output -raw propaganda | base64 -d
```

3.5. Types / Functions

Preparation

Create a new directory for this exercise:

```
mkdir -p $LAB_ROOT/basics/types
cd $LAB_ROOT/basics/types
```

Documentation for the built-in functions can be found at:
https://www.terraform.io/docs/language/functions/index.html

Step 3.5.1: String interpolation

Create a new file named `strings.tf` and add the following content:

```
locals {
  counter = 5
}

output "counter" {
  value = "Counter is ${local.counter}"
}
```

Run init and apply:

```
terraform init
terraform apply
```

Step 3.5.2: Working with lists

Create a new file named `lists.tf` and add the following content:

```
locals {
  fibonacci = [0,1,1,2,3,5,8,13]
}

output "element_5" {
  value = local.fibonacci.5 // or local.fibonacci[5]
}

output "fibonacci" {
  value = join("/", local.fibonacci)
}
```

Run apply:

```
terraform apply
```

Step 3.5.3: Working with maps

Create a new file named `maps.tf` and add the following content:

```
locals {
  tags = {
    env = "prod"
    app = "nginx"
  }
  extra_tags = {
    platform = "azure"
  }
}

output "tag_list" {
  value = keys(local.tags)
}

output "full_tags" {
  value = merge(local.tags, local.extra_tags)
}
```

Run apply:

```
terraform apply
```

Step 3.5.4: Working with external YAML/JSON files

Terraform provides built-in functions to access external YAML and JSON files.

Create a new file named `project.yaml` and add the following content:

```
components:
  - name: "project-name"
    metadata:
      annotations:
        app: "example"
```

Create a new file named `yaml.tf` and add the following content:

```
locals {
  yaml_file  = yamldecode(file("project.yaml"))
}

output "app" {
  value = local.yaml_file.components.0.metadata.annotations.app
}
```

- acend gmbh

The example above could also be shortened using output chaining to the following snippet but readability suffers:

```
output "app2" {
  value = yamldecode(file("project.yaml")).components.0.metadata.annotations.app
}
```

Run apply:

```
terraform apply
```

Explanation

The statement

```
locals {
  yaml_file  = yamldecode(file("project.yaml"))
}
```

loads the file `project.yaml` and assigns it to the local variable `yaml_file`.
The statement

```
output "app" {
  value = local.yaml_file.components.0.metadata.annotations.app
}
```

creates an output variable, whereas the part `components.0.metadata.annotations.app` refers to the YAML structure

```
components:
  - name: "project-name"
    metadata:
      annotations:
        app: "example"
```

The `components` is a list of which we take the first (0th) element and access sub-attributes.

- acend gmbh

4. Intermediate

We will learn lock Terraform versions and use different configurations to keep our code DRY.

4.1. Versions

Preparation

Finish the *Data Sources exercise* and copy the directory:

```
mkdir -p $LAB_ROOT/intermediate/
cp -r $LAB_ROOT/basics/data_sources $LAB_ROOT/intermediate/versions
cd $LAB_ROOT/intermediate/versions
```

Step 4.1.1: Create versions.tf

Create a new file named `versions.tf` and add the following content:

```
terraform {
  required_version = "= 1.11.2"

  required_providers {
    random  = {
      source  = "hashicorp/random"
      version = "= 3.7.1"
    }
    local   = {
      source  = "hashicorp/local"
      version = "= 2.5.2"
    }
  }
}
```

Pin the `required_version` to the Terraform version you are using localy!

Explanation

With multiple engineers working on the same infrastructure code base, it is inevitable to have different versions of the Terraform CLI installed.

Furthermore, are Terraform providers under heavy development and have new features added daily. This rapid development can lead to incompatibilities and trigger regressions; neither are desirable in a production environment

It is best practice to lock the Terraform CLI and provider versions to a specific release. This ensures a controlled version management and planned upgrades.

Step 4.1.2: Init Terraform

Now delete the existing terraform providers and lock files (optional), init the stack and apply it by running:

```
rm -r .terraform/ .terraform.lock.hcl
terraform init
terraform apply
```

Error

If you see any error because on “Unsupported Terraform Core version”, please update the version.tf with the installed verion.

```
terraform version
```

Step 4.1.3: Terraform code formatting

Terraform offers a command to format all files according to HashiCorp guidelines by running the following command:

```
terraform fmt -recursive -diff
```

Note

Most IDEs offer HCL formatting but it differs from the HashiCorp guidelines. It is recommended to use the `terraform fmt` command for compliance.

Note

You can use the `fmt` command of Terraform in CI/CD pipelines to check if the code has been formatted correctly. Use the following command in the root folder of your Terraform code base:

```
terraform fmt -recursive -check
```

4.2. Count / Loops

Preparation

Create a new directory for this exercise:

```
mkdir -p $LAB_ROOT/intermediate/count_loops
cd $LAB_ROOT/intermediate/count_loops
```

Optional: Create empty files:

```
touch {main,elvis,multiple,outputs}.tf
```

Step 4.2.1: Conditional resource

By adding the identifier `count` to a resource, you can either make the resource conditional or create multiple instances.

Create a new file named `elvis.tf` in your working directory and paste the following:

```
locals {
  create_password = false
}

resource "random_password" "optional_password" {
  count  = local.create_password ? 1 : 0
  length = 16
}

output "optional_password" {
  sensitive = true
  value     = local.create_password ? random_password.optional_password.0.result : null
}
```

Explanation

The `count` identifier is (ab)used to create 0 instances of `random_password`. In case multiple instances exist, the resource turns into an array and has to be referenced using the `.0` index.

Step 4.2.2: Multiple resources using `count`

Multiple resources can be instantiated by increasing the `count` value.

Create a new file named `multiple.tf` in your working directory and paste the following:

- acend gmbh

```
resource "random_uuid" "ids" {
  count  = 8
}

output "ids" {
  value = random_uuid.ids.*.result
}
```

The `terraform apply` output will look similar to this:

```
...
Apply complete! Resources: 8 added, 0 changed, 0 destroyed.

Outputs:

ids = [
  "87745fa2-2515-507c-7bde-624d67f31c72",
  "a0cd9772-ab30-3752-b313-ea5b3e82cd49",
  "c6e51356-dd04-3fc2-9d7c-4b222325e92a",
  "4a828a5c-b6fc-d4de-1f07-d2e6511507f3",
  "a75e48ee-9397-d13a-dd94-e26118589156",
  "94efcb57-7981-0ec6-387a-3b01bbab429f",
  "a34be5b3-43f2-e673-7f9d-c7fa6f6e0ef9",
  "9cb5c592-a917-4f21-834d-3eed10a3fba8",
]
```

Explanation

Having `count = 8` creates 8 UUID instances. The wildcard selector `*` can be used to access the `result` attribute of all instances and create a list; see the generated output.

Step 4.2.3: Multiple resources using `for_each`

Multiple resources can also be instantiated by using a `set` or a `map`. The identifier `for_each` is loops over the entries of the collection and exposes the entry of the iteration.

Add the following content to the end of the file `multiple.tf`:

```
locals {
  files = {
    "aws.txt"   = "Jeff Bezos"
    "azure.txt" = "Bill Gates"
    "gcp.txt"   = "Larry Page and Sergey Brin"
  }
}

resource "local_file" "cloud_godfathers" {
  for_each = local.files

  filename = each.key
  content  = each.value
}
```

Explanation

The `for_each` loop sets the `key` and `value` attributes of the iterator `each` according to the map items. This

construct allows the dynamic creation of resources based on a variable.

Step 4.2.4: `for` -loops (list / map comprehension)

List and maps can be iterated using a `for` -loop to modify, extract and/or filter records.

Add the following content to the file `outputs.tf` :

```
locals {
  planets = [
    "mars",
    "saturn",
    "venus"
  ]
}

output "planets" {
  value = [for p in local.planets : title(p)]
}
```

Run `terraform init` followed by `terraform apply` to see the result.

The `map` `for` -loop works very similar, but operates on a key/value pair.
Add the following `map` to `outputs.tf` :

```
locals {
  objects = {
    "mars"   = "planet",
    "saturn" = "planet",
    "venus"  = "planet",
    "sun"    = "star"
  }
}

output "is_star" {
  value = {for k,v in local.objects : k => v == "star"}
}
```

Explanation

The list `for` -loop iterates over all `planets` and upper-cases the first character (aka “title-case”).

The map `for` -loop iterates over all `objects` and prints `true` / `false` if the object is a star.

Try it out

Print a `list` of all objects which are stars. Use the following snippet:

```
output "stars" {
  value = ["todo"]
}
```

Note

You can use `if` statements to filter elements, see:
https://developer.hashicorp.com/terraform/language/expressions/for#filtering-elements

4.3. Backend State

Preparation

Create a new directory for this exercise:

```
mkdir -p $LAB_ROOT/intermediate/backend_state
cd $LAB_ROOT/intermediate/backend_state
```

Optional: Create empty files:

```
touch main.tf
```

Step 4.3.1: Define a backend

Create a new file named `main.tf` and add the following content:

```
terraform {
  backend "local" {
    path = "foobar.tfstate"
  }
}

resource "random_password" "super_secret" {
  length = 16
}
```

Run the commands

```
terraform init
terraform apply
```

After the apply run:

```
ls -al
```

Now you should see a local file named `foobar.tfstate` containing the Terraform state.

Step 4.3.2: List all managed resources

Terraform has builtin commands to interact with the state.

Run the following command to list all managed resources:

```
terraform state list
```

Run the following command to show a specific resource in the state:

```
terraform state show random_password.super_secret
```

Advanced: Run the following command to fetch the raw JSON terraform state:

```
terraform state pull
```

> **Note**
>
> The password in the JSON field `"result"` is stored in clear text! That’s why the Terraform state file should be considered sensitive and protected accordingly!

4.4. Config Files

Preparation

Create a new directory for this exercise:

```
mkdir -p $LAB_ROOT/intermediate/multi_env
cd $LAB_ROOT/intermediate/multi_env
```

Optional: Create empty files:

```
touch {main,variables,outputs}.tf
```

Step 4.4.1: Define variable and output

Create a new file named `variables.tf` and add the following content:

```
variable "environment" {}
```

Create a new file named `outputs.tf` and add the following content:

```
output "current_env" {
  value = var.environment
}
```

Create a new file named `main.tf` and add the following content:

```
terraform {
  backend "local" {}
}
```

Explanation

The backend of type `local` is declared but missing the `path` argument; this is a so-called “partial configuration”. The missing argument will be added via a config file.

Step 4.4.2: Offload configuration to separate files

It is best practice separating configuration from HCL code. For this purpose we create a dedicated directory:

```
mkdir config
```

Create a new file named `config/dev.tfvars` and add the following content:

```
environment = "dev"
```

Create a new file named `config/dev_backend.tfvars` and add the following content:

```
path = "dev.tfstate"
```

Step 4.4.3: Init and apply using config files

Now we init Terraform by specifying a backend configuration with the option `-backend-config`:

```
terraform init -backend-config=config/dev_backend.tfvars
```

Then we apply the code by specifying a variable configuration with the option `-var-file`:

```
terraform apply -var-file=config/dev.tfvars
```

You should now see the following output:

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

Outputs:

current_env = "dev"
```

And a state file called `dev.tfstate` containing the Terraform state.

Explanation

The backend and variable configuration files abstract the code from different “instances”. This pattern can be used to provision different environments like dev, test, prod.

Step 4.4.4: Create a production configuration

To add another set of configuration for a “production” environment, lets just add two more files:

Create a new file named `config/prod.tfvars` and add the following content:

```
environment = "prod"
```

Create a new file named `config/prod_backend.tfvars` and add the following content:

```
path = "prod.tfstate"
```

> **Warning**
> We need to re-initialize Terraform to use the new state by providing the argument `-reconfigure` (or by deleting the `.terraform` directory) and then run the usual apply.

```
terraform init -backend-config=config/prod_backend.tfvars -reconfigure
terraform apply -var-file=config/prod.tfvars
```

You should now see two Terraform state files for each set of configuration:

- `dev.tfstate`
- `prod.tfstate`

> **Note**
> The separation of configuration in the `config/` directory keeps the HCL code DRY.
> It is a common pattern to have many different environments or customer configurations in this directory, which shall be under source control.

> **Warning**
> Do NOT store any sensitive information like credentials or keys in the configuration!
> Use a secrets management system like HashiCorp Vault, AWS SecretsManager, 1Password etc

Try it out

It is a common pattern to set credentials via the shell environment. Terraform has built-in support to set variables via environment by prefixing the Terraform variable name with `TF_VAR_` .

Add the following to `variables.tf` :

```
variable "secret" { }
```

and the following to `outputs.tf` :

```
output "secret" {
  value = var.secret
}
```

Then set the value in the shell:

```
export TF_VAR_secret=mysupersecret
```

Now run `terraform apply -var-file=config/prod.tfvars`

5. Advanced

We will learn how to implement modules and explore advanced features of Terraform.

5.1. Modules

Preparation

Create a new directory for this exercise:

```
mkdir -p $LAB_ROOT/advanced/modules
cd $LAB_ROOT/advanced/modules
```

Optional: Create empty file:

```
touch main.tf
```

Step 5.1.1: Define the module

A local module resides in its own directory, lets create one be running:

```
mkdir random_file
```

Create a new file named `random_file/variables.tf` and add the following content:

```
variable "extension" {}
variable "size" {}
```

Create a new file named `random_file/main.tf` and add the following content:

```
resource "random_pet" "filename" { }

resource "random_password" "content" {
  length = var.size
}

resource "local_file" "this" {
  filename = "${random_pet.filename.id}.${var.extension}"
  content = random_password.content.result
}
```

Create a new file named `random_file/outputs.tf` and add the following content:

```
output "filename" {
  value = local_file.this.filename
}
```

Explanation

It is common practice implementing a module with these three files:

- `main.tf`
- `variables.tf`
- `outputs.tf`

For modules with many resouces (10+), it is advised to split `main.tf` into groups of resources.

Step 5.1.2: Create two instances of the module

Create a new file named `main.tf` and add the following content:

```
module "first" {
  source    = "./random_file"
  extension = "txt"
  size      = 1337
}

module "second" {
  source    = "./random_file"
  extension = "txt"
  size      = 42
}

output "filenames" {
  value = [
    module.first.filename,
    module.second.filename
  ]
}
```

Now run

```
terraform init
terraform apply
```

Explanation

We instantiate the `random_file` module two times and specify different parameters. The output `filenames` prints the randomly generated filenames.

5.2. Meta-Arguments

Preparation

Create a new directory for this exercise:

```
mkdir -p $LAB_ROOT/advanced/meta_arguments
cd $LAB_ROOT/advanced/meta_arguments
```

Optional: Create empty files:

```
touch main.tf
```

Step 5.2.1: Missing dependency

Sometimes Terraform can not imply the dependency between resources explicitly. For such cases, a dependency is added to one or multiple resources or data sources. Consider the following snippets.

Create a new file named `main.tf` and add the following content:

```
resource "local_file" "foobar_txt" {
  content  = "4thelulz"
  filename = "foobar.txt"
}

data "local_file" "reference" {
  filename = "foobar.txt"
}
```

Now run:

```
terraform init
terraform apply
```

This will print the following error:

```
Error: open foobar.txt: no such file or directory

  with data.local_file.reference,
  on main.tf line 5, in data "local_file" "foobar_txt":
   5: data "local_file" "reference" {
```

- acend gmbh

Explanation

The data source `local_file.reference` is refreshed at the execution of `terraform apply`. However at this stage, the file does not exist yet and Terraform fails.

Step 5.2.2: Explicit dependency

Change the resource `local_file.reference` as followed:

```
data "local_file" "reference" {
  filename = "foobar.txt"

  depends_on = [local_file.foobar_txt]
}
```

Now run:

```
terraform init
terraform apply
```

Terraform will skip trying to refresh (access) `local_file.reference` because of the explicit dependency on the resource `local_file.foobar_txt` which does not yet exist.

Step 5.2.3: Ignoring external changes

We set the file content to be `4thelulz`. Now lets change it and run apply again:

```
echo 4real > foobar.txt
terraform apply
```

Terraform will restore the file `foobar.txt` to the configuration defined in the code. All good!

But sometimes we don't want that behaviour - we want to ignore the content.
Luckily Terraform offers another meta-argument for this purpose.

Change the `data` `local_file.foobar_txt` as followed:

```
resource "local_file" "foobar_txt" {
  content  = "4thelulz"
  filename = "foobar.txt"

  lifecycle {
    ignore_changes = [content]
  }
}
```

Note

The `content` has changed!

Now run:

```
terraform apply
```

And Terraform will happily ignore the `content = "4thelulz"` .

Explanation

This is particularly useful in cloud engineering to set initial values for tags or secrets and expect an external system or user to override or extend the value.

5.3. Various

Preparation

Create a new directory for this exercise:

```
mkdir $LAB_ROOT/advanced/various
cd $LAB_ROOT/advanced/various
```

Optional: Create empty files:

```
touch {main,variables,outputs}.tf
```

Step 5.3.1: Variable structure

Terraform variables support nested complex types like nested maps and sets. The `type` keyword of the `variable` block allows the definition of type constraints to enforce the correctness of the input (or default) value. See https://developer.hashicorp.com/terraform/language/expressions/type-constraints for the specification.

Create a new file named `variables.tf` and add the following content:

```
variable "clouds" {
  default = {
    aws = {
      company = "Amazon"
      founder = "Jeff Bezos"
      cloud_rank = 1
    }
    azure = {
      company = "Microsoft"
      founder = "Bill Gates"
      cloud_rank = 2
    }
    gcp = {
      company = "Google"
      founder = "Larry Page and Sergey Brin"
      cloud_rank = 3
    }
  }
  type = map(object({
    company = string
    founder = string
    cloud_rank = number
  }))
}
```

The code snippet above defines a map for the top three cloud platforms with three attributes:

- company
- founder
- cloud_rank

- acend gmbh

Try it out

Create a list of the `founder` attributes of all `clouds` using a **SINGLE** output using the following snippet:

```
output "founders" {
  value = ["todo"]
}
```

Step 5.3.2: Variable optional and default fields

Defining variables as objects with attributes is very useful, but sometimes we don't want to specify all attributes but use some defaults. This can be achieved by the `optional` keyword.

Add the following snippet to `outputs.tf` :

```
variable "kubernetes" {
  type = object({
    version     = optional(string)
    node_count  = optional(number, 3)
    vm_type     = optional(string, "t3.small")
  })
  default = {
    version = "1.25.5"
  }
}

output "kubernetes" {
  value = var.kubernetes
}
```

When you run `terraform apply` you should see a fully defined `kubernetes` variable:

```
kubernetes = {
  "node_count" = 3
  "version" = "1.25.5"
  "vm_type" = "t3.small"
}
```

Partial initialization of variables is very useful in combination with `config/*.tfvars` files, to only specify the explicit and override values - keeping the config small and tidy!

Step 5.3.3: Variable validation

Sometimes you want to validate if a variable meets certain conditions. For this purpose, the `validation` block can be added to a variable.

Modify `outputs.tf` as followed:

- acend gmbh

```
variable "kubernetes" {
  type = object({
    version    = optional(string)
    node_count = optional(number, 0)
    vm_type    = optional(string, "t3.small")
  })
  default = {
    version = "1.25.5"
  }
  validation {
    condition     = var.kubernetes.node_count > 0
    error_message = "Minimum Kubernetes nodes is 1"
  }
}
```

Note: Set the `node_count` default to 0 to trigger a validation error!

Now run `terraform apply` and verify the validation error is printed.

Step 5.3.4: Dynamic blocks

Some Terraform resources (and data sources) have repetitive blocks, for example `archive_file`. See documentation at https://registry.terraform.io/providers/hashicorp/archive/latest/docs/data-sources/file

Example:

```
data "archive_file" "dotfiles" {
  type        = "zip"
  output_path = "dotfiles.zip"

  source {
    content  = "# nothing"
    filename = ".vimrc"
  }

  source {
    content  = "# comment"
    filename = ".ssh/config"
  }
}
```

To add such blocks repetitively, we can use the `dynamic` keyword as documented here: https://www.terraform.io/docs/language/expressions/dynamic-blocks.html

Create a new file named `main.tf` and add the following content:

```
data "archive_file" "clouds" {
  type        = "zip"
  output_path = "clouds.zip"

  dynamic "source" {
    for_each = var.clouds
    content {
      filename = "${source.key}.txt"
      content = jsonencode(source.value)
    }
  }
}
```

This will create a zip file containing a text file for each entry in the `clouds` map variable defined previously.

Now run:

```
terraform init
terraform apply
unzip clouds.zip
cat *txt
```

5.4. Templates

Preparation

Create a new directory for this exercise:

```
mkdir $LAB_ROOT/advanced/templates
cd $LAB_ROOT/advanced/templates
```

Optional: Create empty files:

```
touch {main,variables,outputs}.tf
```

Step 5.4.1: Multiline strings

Sometimes you'd like to construct multiline strings while avoiding `\n` escape sequences for readability. Terraform offers so called "heredoc" style string literals to achieve that. The full documentation can be found at https://www.terraform.io/docs/language/expressions/strings.html

Create a new file named `variables.tf` and add the following content:

```
variable "action" {
  default = "fun"
}
```

Create a new file named `outputs.tf` and add the following content:

```
output "multiline_ugly" {
  value = <<EOT
Cloud
engineering
for ${var.action}!
EOT
}
```

This looks pretty ugly but does the job; create a multiline string.
To add indentation, use the sequence `<<-` to improve readability:

```
output "multiline_pretty" {
  value = <<-EOT
          Cloud
          engineering
          for ${var.action}!
          EOT
}
```

- acend gmbh

Now run:

```
terraform init
terraform apply
terraform output -raw multiline_ugly
terraform output -raw multiline_pretty
```

Step 5.4.2: Template files

Templates can be rather large (ie. firewall config or cloud-init scripts) and bloat the Terraform code. For such use-cases the template is stored in a separate file and sourced using the `templatefile` function documented at https://www.terraform.io/docs/language/functions/templatefile.html

In this real-world example, we will use a cloud-init template that is used for Gitlab runner deployments.

Create a new file named `cloud_init.yml.tpl` and add the following content:

```
#cloud-config
package_upgrade: true
packages:
  - docker.io
write_files:
  - path: /etc/cron.d/cleanup_docker_images
    owner: root:root
    content: |
      0 22 * * * root docker system prune --volumes --force --all >/dev/null 2>&1
  - path: /etc/docker/daemon.json
    owner: root:root
    content: |
      { "data-root": "/mnt/docker" }
runcmd:
  - wget -O /usr/local/bin/gitlab-runner https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner
-linux-amd64
  - chmod +x /usr/local/bin/gitlab-runner
  - useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash
  - /usr/local/bin/gitlab-runner install --user=gitlab-runner --working-directory=/home/gitlab-runner
  - /usr/local/bin/gitlab-runner start
  - /usr/local/bin/gitlab-runner register
      --non-interactive
      --executor docker
      --docker-privileged
      --docker-image docker:latest
      --name ${gitlab_runner_id}
      --url ${gitlab_url}
      --registration-token ${gitlab_runner_token}
      --docker-volumes "/certs/client"
%{ if gitlab_tag_list != null ~}
      --tag-list ${gitlab_tag_list}
%{ endif ~}
```

As you can see, the template contains several variables and supports conditional expressions (if / endif) and for-loops.

In `outputs.tf` add the following output:

- acend gmbh

```
output "cloud_init" {
  value = templatefile("cloud_init.yml.tpl", {
      gitlab_runner_id    = 1
      gitlab_url          = "https://foobar.com"
      gitlab_runner_token = "supersecret"
      gitlab_tag_list     = "linux,highmem"
    })
}
```

Now run:

```
terraform apply
terraform output -raw cloud_init
```

Step 5.4.3: Bonus: Cloud-init output

Cloud-init scripts passed as user-data on cloud platforms while provisioning a new VM, have a max size of 16kb. This is almost always enough, but it is good practice to zip and base64 encode the content. Terraform offers a data source to simplify this process, `template_cloudinit_config` documented at https://registry.terraform.io/providers/hashicorp/template/latest/docs/data-sources/cloudinit_config

Create a new file named `main.tf` and add the following content:

```
data "template_cloudinit_config" "runner" {
  gzip          = true
  base64_encode = true

  part {
    content_type = "text/cloud-config"
    content = templatefile("cloud_init.yml.tpl", {
      gitlab_runner_id    = 1
      gitlab_url          = "https://foobar.com"
      gitlab_runner_token = "supersecret"
      gitlab_tag_list     = "linux,highmem"
    })
  }
}
```

In `outputs.tf` add the following output:

```
output "user_data" {
  value = data.template_cloudinit_config.runner.rendered
}
```

Now run:

```
terraform init
terraform apply
terraform output -raw user_data | base64 -d | gunzip -
```

6. Azure Workshop

We will learn how to configure the Azure provider and provision resources in a subscription.

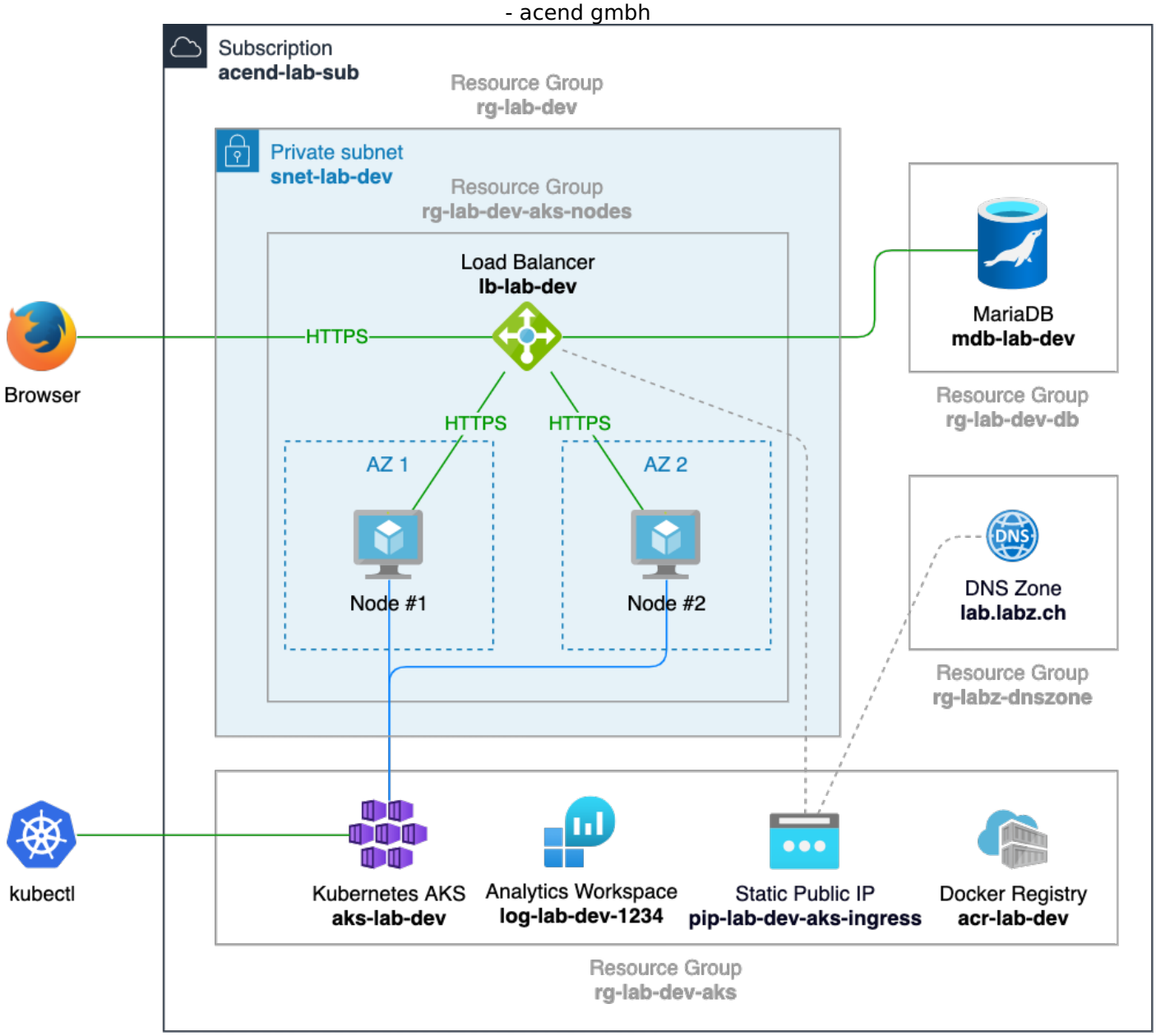
Solution Architecture

Goal: Deploy a HTTPS application on Kubernetes which writes to a database

Main components:

- Kubernetes Cluster (AKS, Azure Kubernetes Service)
- Log Analytics Workspace
- Docker container registry (ACR)
- Virtual Network (Vnet) + Subnet
- Static IP + DNS A record
- Load balancer (Layer 4) + Kubernetes Ingress Controller (NGINX)
- SSL Cert Manager
- MySQL

Diagram



Preparation

Let’s start with the creation of a subfolder for all **azure** exercises:

```
mkdir $LAB_ROOT/azure
cd $LAB_ROOT/azure
```

If you don’t have `az` CLI installed yet, navgiate to https://docs.microsoft.com/en-us/cli/azure/install-azure-cli and follow the instructions.

After installation, run

```
az login --use-device-code
```

and follow the console and web browser instructions.

The Azure naming convention and resource abbreviation can be found at https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/ready/azure-best-practices/resource-abbreviations

6.1. Kubernetes / AKS

Step 6.1.1: Provision the first resource

Create a new file named `main.tf` and add the following content:

```
provider "azurerm" {
  subscription_id = var.subscription_id
  features {
    resource_group {
      prevent_deletion_if_contains_resources = false
    }
  }
}

resource "azurerm_resource_group" "default" {
  name     = "rg-${local.infix}"
  location = var.location
}

data "azurerm_subscription" "current" {}
```

Create a new file named `variables.tf` and add the following content:

```
locals {
  infix = "${var.purpose}-${var.environment}"
}

variable "subscription_id" {}
variable "purpose" {}
variable "environment" {}
variable "location" {}
```

Create a new configuration file file named `config/dev.tfvars` and add the following content:

```
subscription_id = "c1b34118-6a8f-4348-88c2-b0b1f7350f04"
purpose         = "YOUR_USERNAME"
environment     = "dev"
location        = "westeurope"
```

Note: Please replace `YOUR_USERNAME` with the username assigned to you for this workshop.

As seen earlier, its good practice to lock the Terraform CLI and provider versions to avoid uncontrolled version upgrades.

- acend gmbh

Create a new file named `versions.tf` and add the following content:

```
terraform {
  required_version = "> 1.12.0"

  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "=4.46.0"
    }
  }
}
```

> **Versions**
> As of this writing, the current version is `1.11.2`. Set the versions to the latest on by using `terraform version`

Now run

```
terraform init
terraform apply -var-file=config/dev.tfvars
```

You can now navigate to the Azure Portal in your browser and see the newly created resource group with the name `rg-YOUR_USERNAME-dev` at https://portal.azure.com/#@acend.onmicrosoft.com/resource/subscriptions/c1b34118-6a8f-4348-88c2-b0b1f7350f04/resourceGroups

Explanation

The `provider` block configures the Azure provider to use a specific subscription id, provided via configuration file. This structure allows provisioning of the same infrastructure in different subscriptions using different configuration files.

The `location` argument defines in which Azure region the resources shall be deployed. We choose `westeurope` which is the Netherlands because it is amongst the cheapest regions in Europe.

Step 6.1.2: Add a virtual network and subnet

```
  end{vnet}s:red->sub(subnet):::red
```

Create a new file named `network.tf` and add the following content:

```
resource "azurerm_virtual_network" "default" {
  name                = "vnet-${local.infix}"
  location            = azurerm_resource_group.default.location
  resource_group_name = azurerm_resource_group.default.name
  address_space       = [var.network_cidrs.vnet]
}

// There can only be one Network Watcher per subscription; uncomment the following block
// for your own Azure subscriptions outside the lab
//resource "azurerm_network_watcher" "default" {
//  name                = "nw-${local.infix}"
//  location            = azurerm_resource_group.default.location
//  resource_group_name = azurerm_resource_group.default.name
//}

resource "azurerm_subnet" "private" {
  name                 = "snet-${local.infix}-private"
  resource_group_name  = azurerm_virtual_network.default.resource_group_name
  virtual_network_name = azurerm_virtual_network.default.name
  address_prefixes     = [var.network_cidrs.subnet]
}
```

Append the following content to the end of `variables.tf`:

```
variable "network_cidrs" {
  default = {
    vnet    = "10.0.0.0/8"
    subnet  = "10.1.0.0/16"
  }
  type = object({
    vnet    = string
    subnet  = string
  })
}
```

Now run

```
terraform apply -var-file=config/dev.tfvars
```

Explanation

A resource of type `azurerm_network_watcher` is automatically created by Azure for each VNet if not explicitly created. We prefer to provision it via Terraform and align the resource name with our naming convention.

The new variable `network_cidrs` keeps the code DRY by moving values to variables. Since it is unlikely to change, we set a default.

The optional `type` identifier or the variable enforces a specific structure, preventing incomplete configuration.

Step 6.1.3: Add an Analytics Workspace

```
    [illegible]
    end(vnet1g-peSub(subnet)
```

We add an Analytics Workspace to capture the Kubernetes logs and metrics.

Create a new file named `aks.tf` and add the following content:

```
resource "azurerm_resource_group" "aks" {
  location = var.location
  name     = "rg-${local.infix}-aks"
}
```

Create a new file named `analytics_workspace.tf` and add the following content:

```
resource "random_string" "log_analytics_workspace" {
  length  = 4
  special = false
  upper   = false
}

resource "azurerm_log_analytics_workspace" "aks" {
  name                = "log-${local.infix}-${random_string.log_analytics_workspace.result}"
  location            = var.location
  resource_group_name = azurerm_resource_group.aks.name
  sku                 = "PerGB2018"
}
```

We added a resource requiring the random provider, run init again:

```
terraform init
terraform apply -var-file=config/dev.tfvars
```

Explanation

The workspace requires a global unique name. To achieve this, we add a random generated 4-digit number to the workspace name.

Step 6.1.4: Add the Kubernetes cluster (AKS)

```
[illegible]
```

Create a new file named `iam.tf` and add the following content:

```
data "azuread_group" "aks_admins" {
  display_name = var.aks.ad_admin_group
}

resource "azurerm_role_assignment" "students" {
  scope                = azurerm_kubernetes_cluster.aks.id
  role_definition_name = "Azure Kubernetes Service RBAC Cluster Admin"
  principal_id         = data.azuread_group.aks_admins.object_id
}
```

Add the following content to the end of `aks.tf` :

```
resource "azurerm_kubernetes_cluster" "aks" {
  name                = "aks-${local.infix}"
  location            = var.location
  resource_group_name = azurerm_resource_group.aks.name
  node_resource_group = "${azurerm_resource_group.aks.name}-nodes"
  dns_prefix          = local.infix
  kubernetes_version  = var.aks.kubernetes_version

  default_node_pool {
    name                = "linux"
    type                = "VirtualMachineScaleSets"
    vnet_subnet_id      = azurerm_subnet.private.id
    vm_size             = var.aks.node_pool.vm_size
    node_count          = var.aks.node_pool.node_count

    upgrade_settings {
      drain_timeout_in_minutes      = 0
      max_surge                     = "10%"
      node_soak_duration_in_minutes = 0
    }
  }

  network_profile {
    network_plugin    = "kubenet"
    load_balancer_sku = "standard"
  }

  identity {
    type = "SystemAssigned"
  }

  role_based_access_control_enabled = true
  azure_active_directory_role_based_access_control {
    tenant_id          = data.azurerm_subscription.current.tenant_id
    azure_rbac_enabled = true
  }

  oms_agent {
    log_analytics_workspace_id = azurerm_log_analytics_workspace.aks.id
  }
}

resource "azurerm_role_assignment" "aks_identity_monitoring" {
  scope                = azurerm_kubernetes_cluster.aks.id
  role_definition_name = "Monitoring Metrics Publisher"
  principal_id         = azurerm_kubernetes_cluster.aks.oms_agent[0].oms_agent_identity[0].object_id
}

resource "azurerm_role_assignment" "aks_identity_networking" {
  scope                = data.azurerm_subscription.current.id
  role_definition_name = "Network Contributor"
  principal_id         = azurerm_kubernetes_cluster.aks.identity[0].principal_id
}
```

Add the following content to the end of `variables.tf` :

```
variable "aks" {
  type = object({
    kubernetes_version    = string
    log_retention_in_days = number
    ad_admin_group        = string
    node_pool = object({
      vm_size    = string
      node_count = number
    })
  })
}
```

Add the following content to the end of `config/dev.tfvars` (check the latest kubernetes version and use it as input):

```
aks = {
  // az aks get-versions --location westeurope -o table
  kubernetes_version    = "1.33.3"
  log_retention_in_days = 30
  ad_admin_group        = "students"
  node_pool = {
    node_count = 2
    vm_size    = "Standard_B2ms"
  }
}
```

Now run

```
terraform apply -var-file=config/dev.tfvars
```

Explanation

We provision an AKS cluster with two nodes of type `Standard_B2ms` in different availability zones. The available Kubernetes versions on Azure can be listed by running:

```
az aks get-versions --location westeurope -o table
```

The `identity` (service principal) is managed by Azure and the monitoring agent `oms_agent` is configured to use the newly created analytics workspace.

The `azurerm_role_assignment` resources grant roles to the AKS identity;

- `Monitoring Metrics Publisher` allows AKS to publish to the analytics workspace
- `Network Contributor` allows AKS to provision a layer 4 load balancer

Step 6.1.5: Add a Docker registry (ACR)

Create a new file named `acr.tf` and add the following content:

```
resource "random_integer" "acr" {
  min = 1000
  max = 9999
}

resource "azurerm_container_registry" "aks" {
  name                = "cr${replace(local.infix, "-", "")}${random_integer.acr.result}"
  location            = var.location
  resource_group_name = azurerm_resource_group.aks.name
  admin_enabled       = true
  sku                 = "Basic"
}
```

Add the following content to the end of `aks.tf`:

```
resource "azurerm_role_assignment" "aks_identity_acr" {
  scope                = azurerm_container_registry.aks.id
  role_definition_name = "AcrPull"
  principal_id         = azurerm_kubernetes_cluster.aks.kubelet_identity.0.object_id
}
```

Now run

```
terraform init
terraform apply -var-file=config/dev.tfvars
```

Explanation

Similar to the analytics workspace, the Docker container registry has to have a global unique name and cannot contain non-alphanumeric characters; hence `-` are not allowed. We append a random 4-digit number to the ACR name.

We grant the AKS identity the `AcrPull` role to allow AKS to pull Docker images from the registry without providing explicit credentials.

6.2. Remote State

Step 6.2.1: Create a storage

The Azure storage account and storage container to store the Terraform state are not managed by Terraform; it is a chicken and egg problem we resolve by using the `az` CLI as followed:

```
export NAME=YOUR_USERNAME
export ACCOUNT=tfstate$RANDOM
```

```
az group create --location westeurope --name rg-terraform-$NAME
az storage account create --name $ACCOUNT --resource-group rg-terraform-$NAME
az storage container create --resource-group rg-terraform-$NAME --account-name $ACCOUNT --name terraform-state --public
-access off --auth-mode login
echo $ACCOUNT
```

Note: Please replace `YOUR_USERNAME` with the username assigned to you for this workshop.

> **Important**
>
> `YOUR_USERNAME` for this and all upcoming exercises must be
>
> - all lowercase
> - only contain alpha-numeric characters `^[a-z0-9]$`

Step 6.2.2: Configure the Terraform backend

Add the following content to the **start** of `main.tf` :

```
terraform {
  backend "azurerm" {}
}
```

Create a new file named `config/dev_backend.tfvars` and add the following content:

```
subscription_id      = "c1b34118-6a8f-4348-88c2-b0b1f7350f04"
resource_group_name  = "rg-terraform-YOUR_USERNAME"
storage_account_name = "YOUR_ACCOUNT"
container_name       = "terraform-state"
key                  = "dev.tfstate"
```

Note: Please replace `YOUR_USERNAME` with the username assigned to you for this workshop and `YOUR_ACCOUNT` with the value of the `$ACCOUNT` variable.

Now run

```
terraform init -backend-config=config/dev_backend.tfvars
```

Terraform will detect that a local state already exists and asks if you would like to migrate the local state to the new remote state; enter `yes` :

```
Initializing the backend...
Do you want to copy existing state to the new backend?
  Pre-existing state was found while migrating the previous "local" backend to the
  newly configured "azurerm" backend. No existing state was found in the newly
  configured "azurerm" backend. Do you want to copy this state to the new "azurerm"
  backend? Enter "yes" to copy and "no" to start with an empty state.

  Enter a value: yes
...
```

Explanation

The Azure storage account is another resource which requires a global unique name. We therefore prefix and randomise the name.

6.3. Load Balancer

Step 6.3.1: Create a kubernetes namespace

Add the following content below the existing `provider` block of `main.tf` :

```
provider "kubernetes" {
  host                   = azurerm_kubernetes_cluster.aks.kube_admin_config.0.host
  client_certificate     = base64decode(azurerm_kubernetes_cluster.aks.kube_admin_config.0.client_certificate)
  client_key             = base64decode(azurerm_kubernetes_cluster.aks.kube_admin_config.0.client_key)
  cluster_ca_certificate = base64decode(azurerm_kubernetes_cluster.aks.kube_admin_config.0.cluster_ca_certificate)
}
```

Create a new file named `nginx_ingress.tf` and add the following content:

```
resource "kubernetes_namespace" "nginx_ingress" {
  metadata {
    name = "nginx-ingress"
  }
}
```

Since we added a new provider, Terraform needs to be initialized again:

```
terraform init -backend-config=config/dev_backend.tfvars
terraform apply -var-file=config/dev.tfvars
```

Explanation

We use the Kubernetes provider to create a namespace named `nginx-ingress` . The provider is configured using attributes of the AKS cluster; this a good example demonstrating the power of Terraform to use multiple providers.

Step 6.3.2: Add a public static IP

Add the following content below the `azurerm_resource_group` block in `aks.tf` :

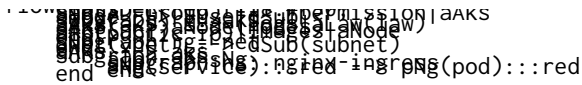
- acend gmbh

```
resource "azurerm_public_ip" "aks_lb_ingress" {
  name                = "pip-${local.infix}-aks-lb-ingress"
  location            = var.location
  resource_group_name = azurerm_resource_group.aks.name
  allocation_method   = "Static"
  sku                 = "Standard"
}
```

Now run

```
terraform apply -var-file=config/dev.tfvars
```

Step 6.3.3: Install NGINX ingress controller



Add the following content below the existing Kubernetes `provider` block of `main.tf`:

```
provider "helm" {
  kubernetes = {
    host                   = azurerm_kubernetes_cluster.aks.kube_admin_config.0.host
    client_certificate     = base64decode(azurerm_kubernetes_cluster.aks.kube_admin_config.0.client_certificate)
    client_key             = base64decode(azurerm_kubernetes_cluster.aks.kube_admin_config.0.client_key)
    cluster_ca_certificate = base64decode(azurerm_kubernetes_cluster.aks.kube_admin_config.0.cluster_ca_certificate)
  }
}
```

Add the following content to the end of `nginx_ingress.tf`:

- acend gmbh

```
resource "helm_release" "nginx_ingress" {
  name          = "nginx-ingress"
  namespace     = kubernetes_namespace.nginx_ingress.id
  repository    = "https://kubernetes.github.io/ingress-nginx"
  chart         = "ingress-nginx"
  version       = "4.7.0"
  atomic        = true
  reset_values = true
  timeout       = 900

  values = [
    yamlencode({
      fullnameOverride = "nginx-ingress"
      controller = {
        replicaCount = 1
        service = {
          loadBalancerIP = azurerm_public_ip.aks_lb_ingress.ip_address
          annotations = {
            "service.beta.kubernetes.io/azure-load-balancer-health-probe-request-path" = "/healthz"
            "service.beta.kubernetes.io/azure-load-balancer-resource-group"            = azurerm_public_ip.aks_lb_ingre
ss.resource_group_name
            "service.beta.kubernetes.io/azure-load-balancer-internal"                  = "false"
          }
        }
      }
    })
  ]
}
```

Since we added a new provider, Terraform needs to be initialized again:

```
terraform init -backend-config=config/dev_backend.tfvars
terraform apply -var-file=config/dev.tfvars
```

Bonus

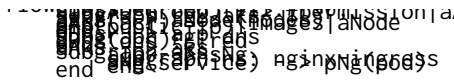
Check the latest version of the helm release here: https://artifacthub.io/packages/helm/ingress-nginx/ingress-nginx and update your terraform file.

Explanation

There is some magic here; Azure AKS will automatically provision a load balancer if the Azure specific service annotations are present. See https://docs.microsoft.com/en-us/azure/aks/ingress-internal-ip for more information.

We set the load balancer IP to the allocated public static IP and deploy a single ingress controller pod; sufficient for this lab.

Step 6.3.4: Configure DNS



Create a new file named `dns.tf` and add the following content:

```
data "azurerm_dns_zone" "parent" {
  name                = "labz.ch"
}

resource "azurerm_dns_zone" "child" {
  name = "${var.purpose}.${data.azurerm_dns_zone.parent.name}"
  resource_group_name = azurerm_resource_group.default.name
}

resource "azurerm_dns_ns_record" "child" {
  name                = var.purpose
  zone_name           = data.azurerm_dns_zone.parent.name
  resource_group_name = data.azurerm_dns_zone.parent.resource_group_name
  ttl                 = 300
  records             = azurerm_dns_zone.child.name_servers
}

resource "azurerm_dns_a_record" "ingress" {
  name                = "*"
  resource_group_name = azurerm_resource_group.default.name
  ttl                 = 300
  zone_name           = azurerm_dns_zone.child.name
  records             = [azurerm_public_ip.aks_lb_ingress.ip_address]
}
```

Now run

```
terraform apply -var-file=config/dev.tfvars
```

Perform a DNS lookup for your subdomain by running:

```
host foobar.YOUR_USERNAME.labz.ch
```

Which should return something like:

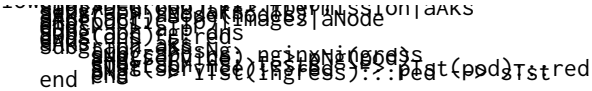
```
foobar.YOUR_USERNAME.labz.ch has address 20.50.15.16
```

Now traffic is ready to be routed to your new Kubernetes cluster!

Explanation

We create a subdomain (child DNS zone in Azure terminology) in the top-level domain `labz.ch` for each workshop participant. The wildcard A record points to the layer 4 load balancer, so all traffic is sent to the load balancer and forwarded to the NGINX ingress controller.

Step 6.3.4: Test HTTP ingress



Before we can deploy workload on Kubernetes, we need to fetch the cluster credentials by running the following command:

```
az aks get-credentials --name aks-YOUR_USERNAME-dev --resource-group rg-YOUR_USERNAME-dev-aks -a
```

Note: Please replace `YOUR_USERNAME` with the username assigned to you for this workshop.

Now check if everything works as expected:

```
kubectl get ns
```

This should show you the following output:

```
NAME              STATUS   AGE
default           Active   3h42m
kube-node-lease   Active   3h42m
kube-public       Active   3h42m
kube-system       Active   3h42m
nginx-ingress     Active   60m
```

Create a new directory for your tests:

```
mkdir tests
```

Create a new file named `tests/http.yaml` and add the following content:

```
# kubectl apply -f http.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: tests

---

apiVersion: v1
kind: Pod
metadata:
  name: hello
  namespace: tests
  labels:
    app: hello
spec:
  containers:
  - image: "nginxdemos/hello:plain-text"
    name: hello
    ports:
    - containerPort: 80
      protocol: TCP

---

apiVersion: v1
kind: Service
metadata:
  name: hello
  namespace: tests
spec:
  selector:
    app: hello
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80

---

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: insecure
  namespace: tests
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  ingressClassName: nginx
  rules:
  - host: insecure.YOUR_USERNAME.labz.ch
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: hello
            port:
              number: 80
```

Note: Please replace `YOUR_USERNAME` with the username assigned to you for this workshop.

Now apply the config by running:

```
kubectl apply -f tests/http.yaml
```

Verify the pod is running:

```
kubectl get pod,ing -n tests
```

This should show the following output:

```
NAME        READY   STATUS    RESTARTS   AGE
hello       1/1     Running   0          97s
```

Now use `curl` to access your service:

```
curl insecure.YOUR_USERNAME.labz.ch
```

This should show the following output:

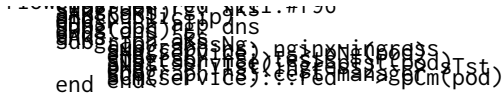
```
Server address: 10.244.0.9:80
Server name: hello
Date: 26/Aug/2021:13:49:10 +0000
URI: /
Request ID: 62c2b4fea5112b355ffe470c3c358817
```

Congratulations! You can now successfully route traffic to your cluster.

- acend gmbh

6.4. SSL Certificates

Step 6.4.1: Cert Manager installation



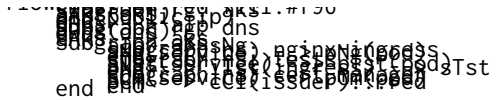
Create a new file named `cert_manager.tf` and add the following content:

```
resource "kubernetes_namespace" "cert_manager" {
  metadata {
    name = "cert-manager"
  }
}

resource "helm_release" "cert_manager" {
  name          = "cert-manager"
  repository    = "https://charts.jetstack.io"
  chart         = "cert-manager"
  version       = "1.18.2"
  namespace     = kubernetes_namespace.cert_manager.id
  atomic        = true
  reset_values = true

  set = [
    {
      name  = "installCRDs"
      value = "true"
    },
    {
      name  = "global.leaderElection.namespace"
      value = kubernetes_namespace.cert_manager.id
    },
    {
        name = "dns01RecursiveNameservers"
        value = "9.9.9.10:53"
    }
  ]
}
```

Now run

```
terraform apply -var-file=config/dev.tfvars
```

Step 6.4.2: Cluster Issuer



Create a new directory structure:

```
mkdir -p helm/cert_manager_issuer/templates
```

Create a new file named `helm/cert_manager_issuer/Chart.yaml` and add the following content:

```
apiVersion: v2
name: cluster-issuers
description: Let's Encrypt cluster issuer for Cert Manager
version: 1.0.0
```

Create a new file named `helm/cert_manager_issuer/templates/cluster_issuer.yaml` and add the following content:

```
# https://cert-manager.io/docs/configuration/acme/dns01/azuredns/
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: noreply@labz.ch
    privateKeySecretRef:
      name: letsencrypt-prod
    solvers:
    - http01:
        ingress:
          class: nginx
```

Add the following content to the end of `cert_manager.tf` :

```
resource "helm_release" "cluster_issuer" {
  name          = "cluster-issuer"
  chart         = "${path.module}/helm/cert_manager_issuer"
  version       = "1.0.0"
  namespace     = kubernetes_namespace.cert_manager.metadata.0.name
  atomic        = true
  reset_values = true

  depends_on = [helm_release.cert_manager]
}
```

Now run

```
terraform apply -var-file=config/dev.tfvars
```

Explanation

Cert Manager needs to be configured with a `ClusterIssuer` Kubernetes resource. This uses a Kubernetes CRD (Custom Resource Definition), which we deploy using a custom Helm chart.

The resource `helm_release.cluster_issuer` depends on `helm_release.cert_manager` because it deploys a CRD which is registered by the Cert Manager, triggering a Kubernetes verification error (API unknown).

Step 6.4.3: Test HTTPS ingress

Create a new file named `tests/https.yaml` and add the following content:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: secure
  namespace: tests
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    cert-manager.io/cluster-issuer: letsencrypt-prod
spec:
  ingressClassName: nginx
  tls:
  - hosts:
    - secure.YOUR_USERNAME.labz.ch
    secretName: tls-secure
  rules:
  - host: secure.YOUR_USERNAME.labz.ch
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: hello
            port:
              number: 80
```

Note: Please replace `YOUR_USERNAME` with the username assigned to you for this workshop.

Now apply the config by running:

```
kubectl apply -f tests/https.yaml
```

Now use `curl` to access your service:

```
curl https://secure.YOUR_USERNAME.labz.ch
```

This should show the following output:

```
Server address: 10.244.0.11:80
Server name: hello
Date: 26/Mar/2025:14:19:30 +0000
URI: /
Request ID: f159d777a93e189c6955f31dde6dba38
```

Congratulations! You can now successfully expose a HTTP service via HTTPS.

6.5. MySQL

Step 6.5.1: Configure AKS egress IP

By default, AKS routes traffic to the internet via a (randomly assigned) Azure public IP. For some scenarios like our MySQL instance, we want to whitelist the source IP to restrict access to the services.

Add the following content below the resource `azurerm_public_ip.aks_lb_ingress` in `aks.tf` :

```
// optional: only needed to control AKS egress IP(s)
resource "azurerm_public_ip" "aks_lb_egress" {
  name                = "pip-${local.infix}-aks-lb-egress"
  location            = var.location
  resource_group_name = azurerm_resource_group.aks.name
  allocation_method   = "Static"
  sku                 = "Standard"
}
```

To configure AKS to use a static egress IP, modify the `azurerm_kubernetes_cluster.aks` resource in `aks.tf` and replace the `network_profile` block with the following content:

```
  network_profile {
    network_plugin    = "kubenet"
    load_balancer_sku = "standard"

    // optional: only needed to control AKS egress IP(s)
    load_balancer_profile {
      outbound_ip_address_ids = [azurerm_public_ip.aks_lb_egress.id]
    }
  }
```

Now run

```
terraform apply -var-file=config/dev.tfvars
```

To verify the egress IP is correct, run the following command:

```
kubectl exec -n tests hello -- curl -s ifconfig.me
```

This lists the egress IP of the AKS cluster as reported by the website https://ifconfig.me

Now verify this IP is equal to the AKS load balancer ip by running:

```
terraform state show azurerm_public_ip.aks_lb_egress
```

Step 6.5.2: Add a MySQL instance

end mariadb(firewall):::red -->mariadb(database):::red

Create a new file named `mysql.tf` and add the following content:

```
resource "azurerm_resource_group" "db" {
  name     = "rg-${local.infix}-db"
  location = var.location
}

resource "random_password" "mysql" {
  length  = 16
  special = false
}

resource "azurerm_mysql_flexible_server" "demo" {
  name                         = "mdb-${local.infix}"
  resource_group_name          = azurerm_resource_group.db.name
  location                     = azurerm_resource_group.db.location
  administrator_login          = "demo"
  administrator_password       = random_password.mysql.result
  sku_name                     = "B_Standard_B1ms"
  backup_retention_days        = 7
  geo_redundant_backup_enabled = false
  version                      = "8.0.21"

  storage {
    size_gb = 20
  }
}

resource "azurerm_mysql_flexible_server_configuration" "ssl_off" {
  name                = "require_secure_transport"
  resource_group_name = azurerm_resource_group.db.name
  server_name         = azurerm_mysql_flexible_server.demo.name
  value               = "off"
}

resource "azurerm_mysql_flexible_database" "demo_app" {
  name                ="demo_app"
  resource_group_name = azurerm_resource_group.db.name
  server_name         = azurerm_mysql_flexible_server.demo.name
  charset             = "utf8"
  collation           = "utf8_general_ci"
}

resource "azurerm_mysql_flexible_server_firewall_rule" "aks_egress_ip" {
  name                = "aks-egress-ip"
  resource_group_name = azurerm_resource_group.db.name
  server_name         = azurerm_mysql_flexible_server.demo.name
  start_ip_address    = azurerm_public_ip.aks_lb_egress.ip_address
  end_ip_address      = azurerm_public_ip.aks_lb_egress.ip_address
}
```

Create a new file named `outputs.tf` and add the following content:

```
output "mysql_uri" {
  sensitive = true
  value     = format("mysql://%s:%s@%s/%s",
    azurerm_mysql_flexible_server.demo.administrator_login,
    azurerm_mysql_flexible_server.demo.administrator_password,
    azurerm_mysql_flexible_server.demo.fqdn,
    azurerm_mysql_flexible_database.demo_app.name
  )
}
```

Now run

```
terraform apply -var-file=config/dev.tfvars
```

The MySQL URI can be displayed by running:

```
terraform output mysql_uri
```

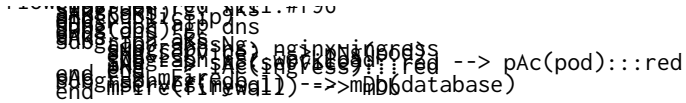
Explanation

The MySQL instance is a managed service by Azure and has a public IP. By default, no IPs are allowed to access the instance. The resource `azurerm_mysql_flexible_server_firewall_rule.aks_egress_ip` adds a firewall rule to whitelist the egress IP of the Kubernetes AKS cluster, which allows apps deployed on the cluster to access MySQL.

To configure our demo app, we need to generate a MySQL URI. The Terraform function `format` has familiar syntax to the GLIBC function `snprintf()` and allows better readable code.

6.6. Demo App

Step 6.6.1: Deploy a workload container



To test the setup end-to-end, we deploy an example application on Kubernetes. The app exposes a web service on port 5000 and writes sample records to the MySQL.

Create a Kubernetes secret containing the MySQL URI to be exposed as the POD environment variable `MYSQL_URI`:

```
kubectl create namespace workload
kubectl create secret generic mysql-uri --namespace workload --from-literal=mysql_uri=$(terraform output -raw mysql_uri
)
```

Create a new file named `tests/workload.yaml` and add the following content:

```
# kubectl apply -f workload.yaml
apiVersion: v1
kind: Pod
metadata:
  name: example
  namespace: workload
  labels:
    app: example
spec:
  containers:
  - image: "quay.io/acend/example-web-python:latest"
    name: example
    ports:
    - containerPort: 5000
      protocol: TCP
    env:
      - name: MYSQL_URI
        valueFrom:
          secretKeyRef:
            name: mysql-uri
            key: mysql_uri

---

apiVersion: v1
kind: Service
metadata:
  name: example
  namespace: workload
spec:
  selector:
    app: example
  ports:
  - protocol: TCP
    port: 5000
    targetPort: 5000

---

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example
  namespace: workload
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    cert-manager.io/cluster-issuer: letsencrypt-prod
spec:
  ingressClassName: nginx
  tls:
  - hosts:
    - workload.YOUR_USERNAME.labz.ch
    secretName: tls-workload
  rules:
  - host: workload.YOUR_USERNAME.labz.ch
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: example
            port:
              number: 5000
```

Note: Please replace `YOUR_USERNAME` with the username assigned to you for this workshop.

Deploy the Kubernetes resources by running:

```
kubectl apply -f tests/workload.yaml
```

The application is now accessible via web browser at https://workload.YOUR_USERNAME.labz.ch

To verify the application is connected to the MySQL, run the following command to inspec the log files:

```
kubectl logs -n workload example | head
```

Step 6.6.2: Optional => rewrite yaml to terraform

There are several solutions for how to deploy workload in Kubernetes. You can either use direct yaml files or Helm, but also Terraform itself by using the kubernetes provider.

Check the following documentation to rewrite the yaml content above and deploy it with terraform as well.

- https://registry.terraform.io/providers/hashicorp/kubernetes/latest/docs/resources/pod_v1
- https://registry.terraform.io/providers/hashicorp/kubernetes/latest/docs/resources/service_v1
- https://registry.terraform.io/providers/hashicorp/kubernetes/latest/docs/resources/ingress_v1

6.7. Container Instances (optional)

Sometimes, you need to run a containerized application outside of your AKS cluster—for example, a monitoring dashboard, health checker, or any lightweight utility.
Or, you might just want to run a single container without the complexity and overhead of managing AKS.

For these scenarios, Azure provides Azure Container Instances (ACI)—a serverless container runtime that allows you to run containers in an isolated, standalone environment with minimal setup.

This lab provides a simple example to demonstrate how to achieve this.

Step 6.7.1: Azure Container Instances

We will create the following:

- Ressource group
- Azure container instance

Create a new folder:

```
mkdir $LAB_ROOT/azure/aci
cd $LAB_ROOT/azure/aci
```

Create a new file named `main.tf` and add the following content:

```
provider "azurerm" {
  subscription_id = var.subscription_id
  features {}
}

resource "azurerm_resource_group" "default" {
  name     = "rg-${var.purpose}-aci"
  location = var.location
}

data "azurerm_subscription" "current" {}
```

Now create the new file named `variables.tf` and add:

```
variable "subscription_id" {
  type        = string
  default     = "c1b34118-6a8f-4348-88c2-b0b1f7350f04"
}
variable "purpose" {
  type        = string
  default     = "YOUR_USERNAME"
}
variable "location" {
  type        = string
  default     = "westeurope"
}
```

Note: Please replace `YOUR_USERNAME` with the username assigned to you for this workshop.

Finally the ACI file named `aci.tf` and add this:

```
resource "azurerm_container_group" "aci" {
  name                = "go-aci-${var.purpose}"
  location            = azurerm_resource_group.default.location
  resource_group_name = azurerm_resource_group.default.name
  ip_address_type     = "Public"
  dns_name_label      = "go-aci-${var.purpose}"
  os_type             = "Linux"

  container {
    name   = "acend-go-example"
    image  = "quay.io/acend/example-web-go:latest"
    cpu    = "0.2"
    memory = "0.2"

    ports {
      port     = 5000
      protocol = "TCP"
    }
  }
}

output "fqdn" {
  value = "http://${azurerm_container_group.aci.fqdn}:5000"
}
```

Deploy the Azure resources by running:

```
terraform init
terraform apply
```

The application is now accessible via web browser at => `terraform output -raw fqdn`

Step 6.7.2: What about security?

As demonstrated, **Azure Container Instances (ACI)** provide a lightweight container runtime with direct access to the exposed container port.
However, if your container doesn't include built-in support for secure endpoints, **you'll need to implement your own solution** to secure the connection.

You might have noticed that the Terraform resource is named `azurerm_container_group`.
This is because it allows you to define **multiple containers within a single container group**.

A common use case is to include a **proxy container** alongside your application to handle secure connections and expose a public or private endpoint.

Your Mission: Secure the Workload

Challenge:
Extend the previous example by **adding an additional container that acts as a reverse proxy** to secure your application endpoint.

If your resources fail, vanish, or behave unexpectedly, the Secretary will disavow all knowledge of your

- acend gmbh

actions.

Useful Resources

- Refer to the Terraform documentation to learn how to define additional containers in a container group.
- Caddy is a powerful and simple solution for securing endpoints with automatic HTTPS.
- Not sure how it all fits together? This blog post provides a practical example of integrating HTTPS with ACI.

If you run into issues, expand the section below for troubleshooting tips:

```
resource "azurerm_container_group" "aci" {
  name                = "go-aci-${var.purpose}"
  location            = azurerm_resource_group.default.location
  resource_group_name = azurerm_resource_group.default.name
  ip_address_type     = "Public"
  dns_name_label      = "go-aci-${var.purpose}"
  os_type             = "Linux"

  container {
    name   = "acend-go-example"
    image  = "quay.io/acend/example-web-go:latest"
    cpu    = "0.2"
    memory = "0.2"
  }

  container {
    name   = "caddy"
    image  = "caddy"
    cpu    = "0.5"
    memory = "0.5"

    ports {
      port     = 443
      protocol = "TCP"
    }

    ports {
      port     = 80
      protocol = "TCP"
    }

    commands = ["caddy", "reverse-proxy", "--from", "go-aci-${var.purpose}.westeurope.azurecontainer.io", "--to", "loca
lhost:5000"]
  }

}

output "fqdn" {
  value = "https://${azurerm_container_group.aci.fqdn}"
}
```

In this setup, it may take a few moments for the certificate to be issued by the provider and become active. To monitor the progress, use the following command to view logs from the Caddy container: `az container logs -g YOUR_RESOURCE_GROUP --name YOUR_CONTAINER_GROUP_NAME --container-name caddy`

If you're considering this approach for a production environment, you should also persist the certificate to external storage. The blog post linked above explains how to implement this.

Do you like this lab? Tell us what you think.

- acend gmbh

7. Self-guided Challenges

Self-guided Challenges

After completing all the labs, try tackling the following challenges from scratch, without relying on existing Terraform code. Each challenge is independent—pick the one that interests you most!

Challenge #1: Upgrade azurerm Provider Version (Entry Level)

In the Azure Workshop, an outdated version of the `azurerm` Terraform provider (v3.117.1) was used. In this challenge, you'll modernize the code base.

Objectives

Using Terraform, complete the following tasks:

- Upgrade the Terraform code to use the latest version of the `azurerm` provider
- Identify any deprecated or removed resources and either **migrate** them to supported alternatives or **re-provision** the components

Challenge #2: GitLab Runner Deployment (Intermediate)

In many CI/CD workflows, it's standard practice to use dedicated GitLab Runners. This challenge guides you through provisioning a configurable number of runners using Terraform.

Objectives

Using Terraform, implement the following:

- Initialize a new Terraform stack at `$LAB_ROOT/gitlab_runner`
- Generate a GitLab Runner token from your GitLab instance (either gitlab.com or a self-hosted GitLab)
  - Runners can be registered at the **group** or **project** level—even for private projects
- Provision a Linux VM configured via **cloud-init**:
  - Register the `gitlab-runner` using the GitLab Runner token
  - Ensure the runner service starts on boot
- Confirm successful registration of the runner in your GitLab group or project settings

Bonus

1. Store the GitLab Runner token securely in **Azure Key Vault** as a secret and reference the secret from the cloud-init template instead sourcing from a variable
2. Create a GitLab CI pipeline in a demo project to verify that the self-hosted Azure runner can execute jobs

Challenge #3: Azure Key Vault + External Secrets Operator (Advanced)

Kubernetes applications often require access to sensitive credentials. Rather than passing them during deployment, this challenge uses **External Secrets Operator** to securely replicate secrets from **Azure Key Vault** into Kubernetes.

Objectives

Using Terraform, implement the following:

- Use the existing “Azure Workshop” Terraform stack at `$LAB_ROOT/azure`
- Create an **Azure Key Vault** instance
- Add a new secret to Key Vault
  - Manually modify the secret later via the Azure Portal
- Configure **AKS OIDC (OpenID Connect)** to enable Federated Identity for workload authentication
- Deploy the **External Secrets Operator** to the AKS cluster
  - Grant permissions to the operator via an **Azure User-Assigned Managed Identity**
- Manually create an `ExternalSecret` custom resource to sync the Key Vault secret into a target Kubernetes namespace

8. Cleanup

To finish the lab and destroy all cloud resources managed by Terraform, please run the following command:

```
cd $LAB_ROOT/azure
terraform destroy -var-file=config/dev.tfvars
az group delete --name rg-terraform-$NAME
cd $LAB_ROOT/azure/aci
terraform destroy
```

Thank you!