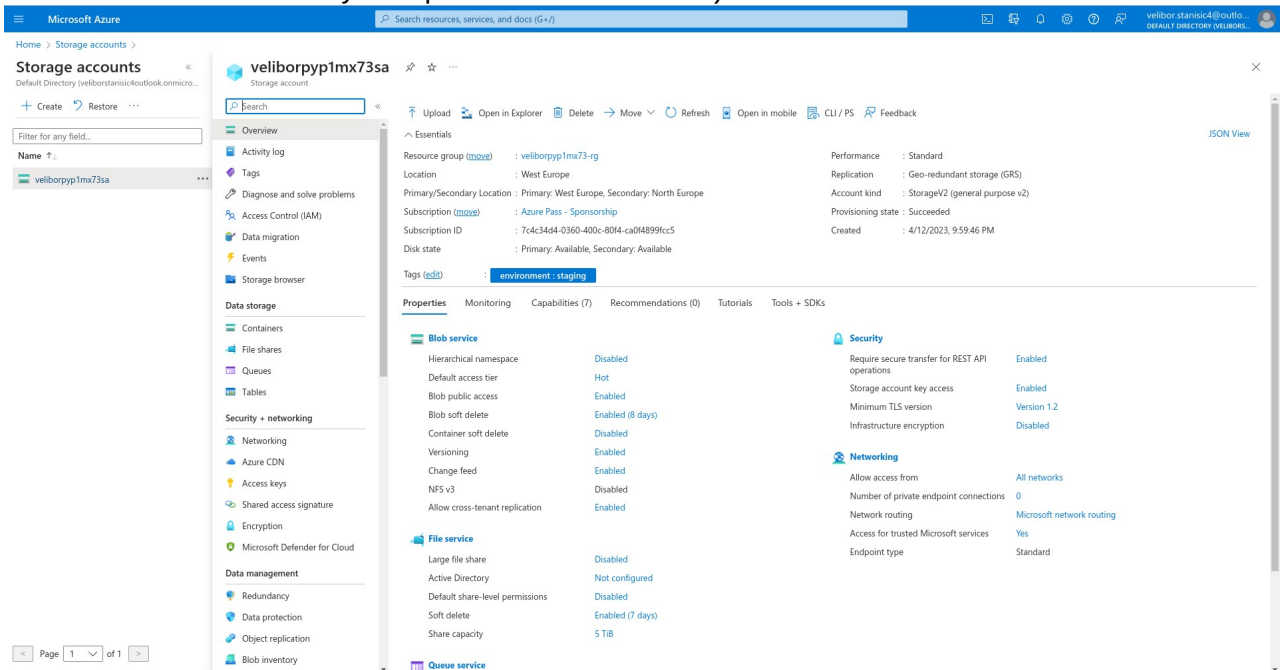


Homework 17 – Velibor Stanisic – Terraform Modules

Task 1: Setup and use remote terraform backend

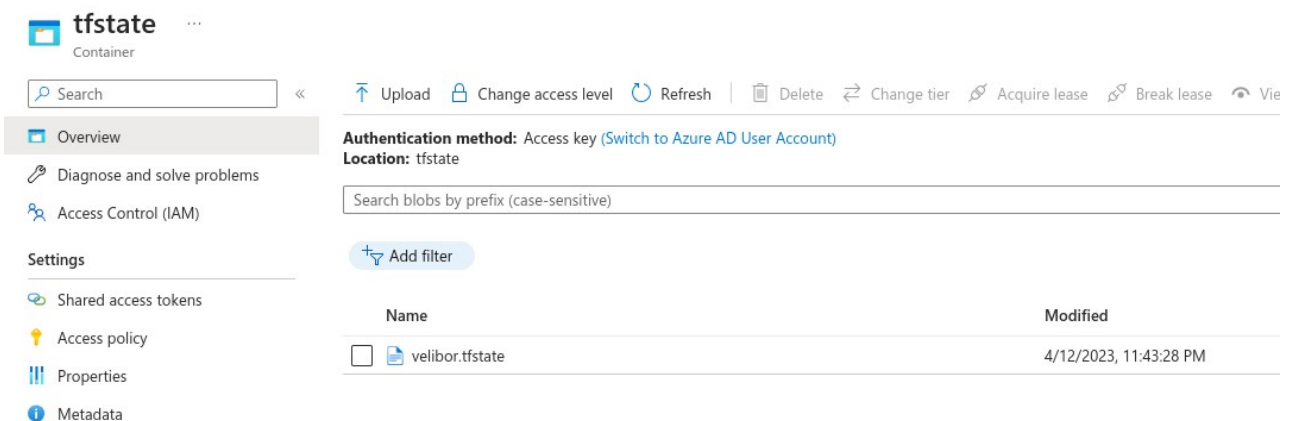
1. In the previous exercise you have created code that creates a storage account. Now we will configure and use that storage account as our backend for terraform. (if you have deleted that storage account, just reapply your terraform code from your previous exercise).



1.1. Create a Container in the storage account named "tfstate" with default settings.

```
[dttw@dttw ~]$ az storage container create --name tfstate --account-name veliborpp1mx73sa --account-key {  
  "created": true  
}  
[dttw@dttw ~]$
```

1.2. Inside the created container, upload an empty file named "`<my_name>.tfstate`"



2. Configure your terraform to use the azurerm remote backend

2.1. Read the terraform documentation for azurerm terraform backend configuration (use authentication using Azure CLI).

2.2. Add the backend configuration to your terraform code.

2.2.1. Create a main.tf file and paste the code snippet from the documentation for backend configuration.

```
main.tf > ...
1 terraform {
2   backend "azurerm" {
3   }
4 }
5
```

2.2.2. Replace the values accordingly:

- resource_group_name - The resource group where the terraform state storage account can be found
- storage_account_name - The storage account name in which the state will be kept
- container_name - The name of the container which will hold the blob with terraform state
- key - the name of the empty file that you uploaded in step 1.2.

```
6 terraform {
7   backend "azurerm" {
8     resource_group_name = "veliborpylmx73sa"
9     storage_account_name = "veliborpylmx73-rg"
10    container_name = "tfstate"
11    key = "velibor.tfstate"
12   }
13 }
14
```

2.3. In case where we have multiple backends configurations for different environments where we configure them in pipelines, we want to be able to switch between different backend configurations for local development also. For this we will use different backend file for each environment.

2.3.1. Create a subdirectory inside your current directory named "backends"

2.3.2. Create a file named <my_name>_env_backend.tf

2.3.3. Move the content from the block - backend "azurerm" in main.tf to the <my_name>_env_backend.tf

2.3.4. The files should have the following content:

- main.tf

```
main.tf > terraform
1 terraform {
2   backend "azurerm" {
3   }
4 }
5
```

- <my_name>_env_backend.tf

```
backends > velibor_env_backend.tf > resource_group_name
1 resource_group_name = "veliborpylmx73-rg"
2 storage_account_name = "veliborpylmx73sa"
3 container_name       = "tfstate"
4 key                  = "velibor.tfstate"
5
```

3. Add the provider configuration for azure without version limitation

3.1. Follow the guidelines from previous exercise about the provider configuration

4. Initialize your terraform code with the azurerm_subscription data source

- Add the following line in your code
data "azurerm_subscription" "current" {}

5. Initialize your terraform code

5.1. Execute the following command

```
terraform init --backend-config=backends/<my_name>_env_backend.tf
```

5.2. The code should initialize successfully and should give you the following output

```
[dttw@dttw tf-exercise-2]$ terraform init --backend-config=backends/velibor_env_backend.tf

Initializing the backend...

Successfully configured the backend "azurerm"! Terraform will automatically
use this backend unless the backend configuration changes.

Initializing provider plugins...
- Finding latest version of hashicorp/azurerm...
- Installing hashicorp/azurerm v3.51.0...
- Installed hashicorp/azurerm v3.51.0 (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!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
[dttw@dttw tf-exercise-2]$
```

6. With this we have finalized our remote backend setup and we can define different backends and switch between them using the command option - backend-config

Task 2: Setup and use remote terraform backend

1. During your midterm assignment we have defined few network resources that were shared and needed to be created before the virtual machine can be deployed and configured and were not owned by the VM itself. Those resources are:

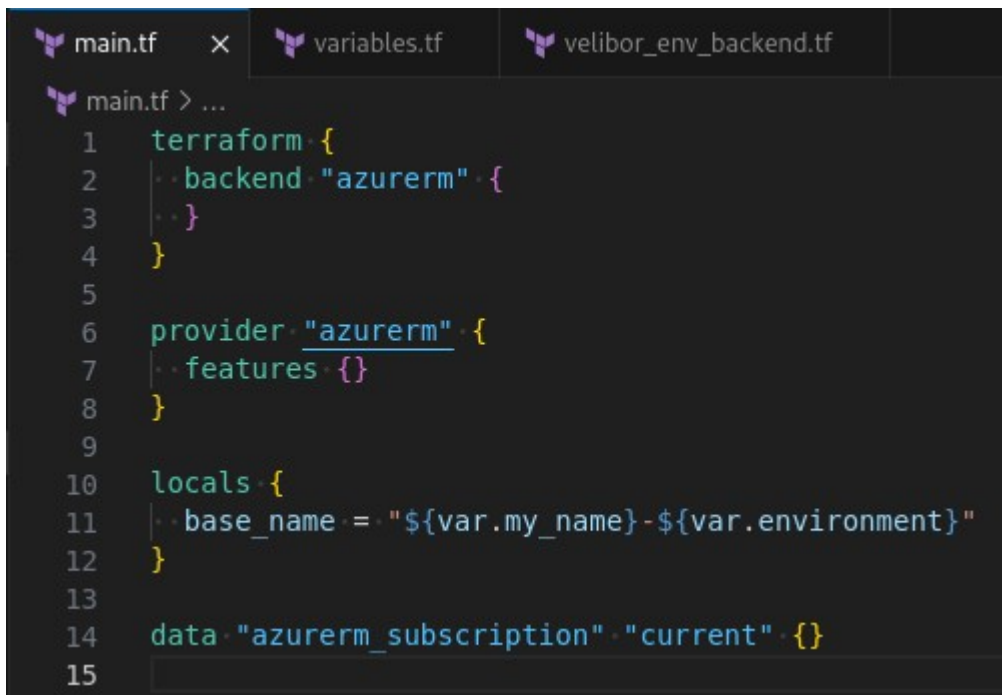
- resource group - base resource used for logical grouping of the resources
- virtual network (VNet) - where the virtual machine will be hosted
- subnet - the subnet where the virtual machine will be deployed

2. Since the resources that will be created are being managed from one terraform code, the simplest way of recognizing that is to use standardized naming for your resources and the simplest grouping of the resources in the code is to use same terraform resource names.

2.1. In the first exercise we have utilized the local values to give standardized names to our resources. Here we will use the same approach.

2.1.1. Define local value named `base_name` with the concatenated values of variables `my_name` and `environment`

```
base_name = "${var.my_name}-${var.environment}"
```



```
main.tf > ...
1  terraform {
2    backend "azurerm" {
3    }
4  }
5
6  provider "azurerm" {
7    features {}
8  }
9
10 locals {
11   base_name = "${var.my_name}-${var.environment}"
12 }
13
14 data "azurerm_subscription" "current" {}
15
```

2.1.2. Declare the variables `my_name` and `environment` in the `variables.tf`



```
variables.tf > ...
1  variable "my_name" {
2    type = string
3  }
4
5  variable "environment" {
6    type = string
7  }
8
```

2.1.3. Define values for the variables in your tfvars file

```
inputs.tfvars > ...
1 my_name = "velibor"
2 environment = "staging"
3 location = "westeurope"
4 |
```

2.1.4. Define network resources network_base_name prefix

network_base_name = "\${local.base_name}-ntwrk"

```
main.tf > ...
1 terraform {
2   backend "azurerm" {
3   }
4 }
5
6 provider "azurerm" {
7   features {}
8 }
9
10 locals {
11   base_name = "${var.my_name}-${var.environment}"
12   network_base_name = "${local.base_name}-ntwrk"
13 }
14
15 data "azurerm_subscription" "current" {}
16 |
```

2.2. Execute terraform plan with the input from your tfvars file

2.2.1. You should not see any errors and your output should be like below. If you have any errors, then you have written something incorrectly in your code. Try to resolve it by reading the error message.

```
data.azurerm_subscription.current: Reading...
data.azurerm_subscription.current: Read complete after 0s [id=/subscriptions/7c4c34d4-0360-400c-80f4-ca0f4899fcc5]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.
```

3. Define the general network resources

3.1. Create a resource group with following parameters (see terraform registry documentation for azurerm_resource_group):

- Terraform resource name - general_network
- name - \${local.network_base_name}-rg
- location - var.location

3.2. Create a virtual network with following parameters (azurerm_virtual_network):

- Terraform resource name - general_network
- name - \${local.network_base_name}-vnet
- location - reference the general_network resource group location attribute

- resource_group_name - reference the general_network resource group name attribute
- address_space - ["10.0.0.0/16"]

3.3. Create a subnet with following parameters (azurerm_subnet):

- Terraform resource name - general_network_vms
- name - \${azurerm_virtual_network.general_network.name}-vms-snet
- resource_group_name - reference the general_network resource group name attribute
- virtual_network_name - reference the general_network virtual network name attribute
- address_prefixes - ["10.0.1.0/24"]

```
resource "azurerm_resource_group" "general_network" {
  name = "${local.network_base_name}-rg"
  location = var.location
}

resource "azurerm_virtual_network" "general_network" {
  name = "${local.network_base_name}-vnet"
  location = azurerm_resource_group.general_network.location
  resource_group_name = azurerm_resource_group.general_network.name
  address_space = ["10.0.0.0/16"]
}

resource "azurerm_subnet" "general_network_vms" {
  name = "${azurerm_virtual_network.general_network.name}-vms-snet"
  resource_group_name = azurerm_resource_group.general_network.name
  virtual_network_name = azurerm_virtual_network.general_network.name
  address_prefixes = ["10.0.1.0/24"]
}
```

3.4. Execute terraform plan with the input from your tfvars file

3.4.1. Your plan should not throw any errors. If any errors found troubleshoot your code from the error information.

```
[dttw@dtw tf-exercise-2]$ terraform plan -var-file=inputs.tfvars
data.azurerm_subscription.current: Reading...
data.azurerm_subscription.current: Read complete after 0s [id=/subscriptions/7c4c34d4-0360-400c-80f4-ca0f4899fcc5]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# azurerm_resource_group.general_network will be created
+ resource "azurerm_resource_group" "general_network" {
  id = (known after apply)
  location = "westeurope"
  name = "velibor-staging-ntwrk-rg"
}

# azurerm_subnet.general_network_vms will be created
+ resource "azurerm_subnet" "general_network_vms" {
  address_prefixes = [
    "10.0.1.0/24",
  ]
  enforce_private_link_endpoint_network_policies = (known after apply)
  enforce_private_link_service_network_policies = (known after apply)
  id = (known after apply)
  name = "velibor-staging-ntwrk-vnet-vms-snet"
  private_endpoint_network_policies_enabled = (known after apply)
  private_link_service_network_policies_enabled = (known after apply)
  resource_group_name = "velibor-staging-ntwrk-rg"
  virtual_network_name = "velibor-staging-ntwrk-vnet"
}

# azurerm_virtual_network.general_network will be created
+ resource "azurerm_virtual_network" "general_network" {
  address_space = [
    "10.0.0.0/16",
  ]
  dns_servers = (known after apply)
  guid = (known after apply)
  id = (known after apply)
  location = "westeurope"
  name = "velibor-staging-ntwrk-vnet"
  resource_group_name = "velibor-staging-ntwrk-rg"
  subnet = (known after apply)
}

Plan: 3 to add, 0 to change, 0 to destroy.
```

Task 3: Define and group the virtual machine and its resources into a module

1. Since we have already defined the base resources that are general and not directly related to the virtual machine, now we can start working on the linux virtual machine declaration. If we look at the resource `azurerm_linux_virtual_machine` in terraform registry, there are some other resources (components) of the virtual machine that need to be declared first and assigned to the virtual machine. Those resources are:

- resource group - the group where the virtual machine will be placed. We will use this resource group for the resto of the VM components also, so we can have clear understanding what belongs and where
- public IP - this is not related directly to our virtual machine but to the network interface that is used by the virtual machine
- network interface - we need to define a network interface before we create a virtual machine
- network security group (NSG) - which will be configured for management and service public
- access by the virtual machine
- assignment of the NSG to the network interface - this is a separate resource in terraform because of the API functionality of the cloud provider

2. Because here we will also use a module to group the resources configuration, the first step for the module creation is the creation of the directory and the basic module configurations.

2.1. In your current directory create a subdirectory named "vm_module".

2.2. Next, inside the directory we will create our required files for terraform code and variables. Inside the directory create files named `main.tf` and `variables.tf`

3. Since the components here are being defined to be used by the VM as resource, we will create one common name which will be the base for the resources.

3.1. Follow the instructions from previous task and define local value name "vm_name" in `main.tf`, that will have the concatenated value of the base name variable and abbreviated with vm (don't forget to declare the variable in `variable.tf`):

```
vm_name = "${var.base_name}-vm"
```

4. Define the resources

4.1. Define the `azurerm_resource_group` resource with following parameters:

- Terraform resource name - vm
- name - `${local.vm_name}-rg`
- location - `var.location`

4.2. Define the `azurerm_public_ip` with the following parameters:

- Terraform resource name - vm
- name - `${local.vm_name}-pip`
- resource_group_name - reference the name attribute of the vm resource group
- location - reference the location attribute of the vm resource group
- allocation_method - Static

4.3. Define the `azurerm_network_interface` resource with following parameters:

- Terraform resource name - `vm`
- `name` - `${local.vm_name}-nic`
- `resource_group_name` - reference the `name` attribute of the `vm` resource group
- `location` - reference the `location` attribute of the `vm` resource group
- under the `ip_configuration` block define the following parameters:
 - `name` - `external`
 - `subnet_id` - `var.vms_subnet_id` (define the variable in the `variables.tf` file)
 - `private_ip_address_allocation` - `Dynamic`
 - `public_ip_address_id` - reference the `id` attribute from the `vm` public ip resource

4.4. Define the `azurerm_network_security_group` resource with the following parameters

- Terraform resource name - `vm`
- `name` - `${azurerm_network_interface.vm.name}-nsg`
- `resource_group_name` - reference the `name` attribute of the `vm` resource group
- `location` - reference the `location` attribute of the `vm` resource group
- define the first `security_rule` block with following parameters
 - `name` - `allow_ssh_from_my_ip`
 - `priority` - `110`
 - `direction` - `Inbound`
 - `access` - `Allow`
 - `protocol` - `Tcp`
 - `destination_port_range` - `22`
 - `source_address_prefix` - `var.my_public_ip` (define the variable in the `variables` file)
 - `destination_address_prefix` - `"*"`
 - `source_port_range` - `"*"`
- define the second `security_rule` block with following parameters
 - `name` - `allow_http_from_my_ip`
 - `priority` - `100`
 - `direction` - `Inbound`
 - `access` - `Allow` o `protocol` - `Tcp`
 - `destination_port_range` - `80`
 - `source_address_prefix` - `var.my_public_ip`
 - `destination_address_prefix` - `"*"`
 - `source_port_range` - `"*"`

4.5. Now we need to associate the network interface and the network security group using the resource called `network_interface_security_group_association` with following parameters:

- Terraform resource name - `vm_nsg_to_vm_nic`
- `network_interface_id` - reference the `id` attribute of the `vm` nic resource
- `network_security_group_id` - reference the `id` attribute of the `vm` nsg resource

4.6. And at the end we came to the point where we need to define the virtual machine resource

`azurerm_linux_virtual_machine` with following parameters:

- Terraform resource name - `web_srv`
- `name` - the local value `vm_name`
- `resource_group_name` - reference the `name` attribute of the `vm` resource group
- `location` - reference the `location` attribute of the `vm` resource group

- size - Standard_ B2s
- admin_username - adminuser
- network_interface_ids - [azurerm_network_interface.vm.id]
- admin_password - var.my_password
- disable_password_authentication - false
- in the os_disk block define the following parameters:
 - caching - ReadWrite
 - storage_account_type - Standard_LRS
 - In the source_image_reference block define the following parameters:
 - publisher - Canonical
 - offer - UbuntuServer
 - sku - 18.04-LTS
 - version - latest
- Since we defined all module resources, now we need to call the module from our root main.tf file
 - Declare the module named vm with source in the directory vm_module:

4.7. Add the rest of the variables that you have defined with their respective values:

- vms_subnet_id - reference the id attribute from the general_network_vms subnet resource
- my_public_ip - define it as input variable in the variables file and assign your public ip in the tfvars file
- my_password - define it as input variable in the variables file and assign value in the tfvars
- file (not recommended in real use case)

4.8. When we add a module to our code, we need to reinitialize our terraform code

4.9. Now we can execute terraform plan with input from tfvars file

4.10. Since we have declared the public ip in the code, we would need to get it as an output from our code

4.10.1. In the vm_module directory create file named outputs.tf

4.10.2. Inside the file define an output value named "vm_public_ip" for the attribute ip_address from the vm public ip resource

4.10.3. Reference the output value as your code output value like bellow:

```
output "vm_public_ip" {
  value = module.vm.vm_public_ip
}
```

4.11. As last execute another terraform plan with input variables from file and it should be without any errors. If you have errors correct them.

5. Apply the terraform code

6. Get the public IP from the output

7. Connect to the vm using ssh with user adminuser. Usually takes around 2 minutes for the VM to be ready.

8. When successfully connected you should see the prompt from the VM. Provide print screen from it.

```
adminuser@velibor-staging-vm:~$ uname -av
Linux velibor-staging-vm 5.4.0-1105-azure #111-18.04.1-Ubuntu SMP Fri Mar 3 22:47:43 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
adminuser@velibor-staging-vm:~$
```

9. Review all of your resources on the subscription and provide print screen of the resource groups.

The screenshot shows the Microsoft Azure portal interface. On the left, the 'Storage accounts' section is active, displaying a list of storage accounts with 'veliborpp1mx73sa' selected. The main pane shows the details for this storage account, including its resource group 'veliborpp1mx73-rg', location 'West Europe', and subscription 'Azure Pass - Sponsorship'. The 'Properties' tab is selected, showing various settings for Blob, File, and Queue services. The 'Security' and 'Networking' sections are also visible, showing settings like 'Require secure transfer for REST API operations' and 'Allow access from'.

10. Take the time to observe the resources that you created and how the code reflects to the resources

11. Once completed, destroy the resources that you created using terraform destroy command with the parameter from input tfvars file (similar to plan and apply)

12. Once the destroy command has completed go with your terminal context to the directory from exercise 1 and execute terraform destroy over there also using the tfvars file input parameter

13. With this you have successfully completed the exercise and cleaned your subscription from the resources created from your code. (There will be a network watcher resource that is being created by azure automatically when we add vnet, and you can delete it manually)