Use Terraform to build a Kubernetes Cluster and Install Jenkins

Table of Contents

Installing Terraform and Building Linux Infrastructure

Install Terraform

We will be running this project on a Windows machine using Azure CLI.

- Download Terraform binary for windows: https://www.terraform.io/downloads.html

- Unzip the Terraform binary and move it to a folder in your systems PATH. In this example we are going to move Terraform to C:\Windows\System32\Terraform. "C:\Windows\System32" is defaulted in windows system PATH. This allows Terraform to be run without knowing and typing the whole path to the file on the command line.

Setup Terraform access to your Azure account

Terraform supports a number of different methods for authenticating to Azure. To enable Terraform to provision resources into Azure, we will create an Azure AD service principal. The service principal grants your Terraform scripts to provision resources in your Azure subscription.

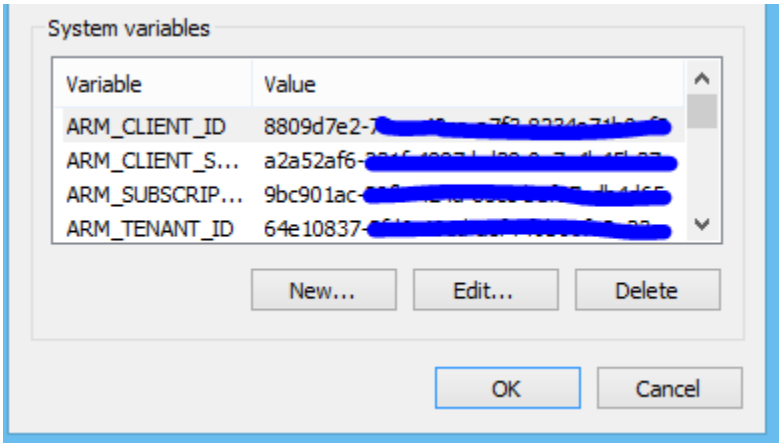
- Login to your Azure account and then click on the Cloud Shell icon on the top right side; this will open a mini shell in the browser.
- On the cloud shell run: `az account list --query "[].{name:name, subscriptionId:id, tenantId:tenantId}"` to get a list of subscription ID and tenant ID values.

```
PS Azure:\> az account list --query "[].{name:name, subscriptionId:id, tenantId:tenantId}"
[
  {
    "name": "Visual Studio Enterprise – MPN",
    "subscriptionId": "9bc901ac-29fb-424d-83cc-bef07adb4d65",
    "tenantId": "64e10837-3fd6-461d-acf4-f9b06fe2a23e"
  }
]
Azure:/
PS Azure:\>
```

- To use a selected subscription, set the subscription for this session with az account set. Set the SUBSCRIPTION_ID environment variable to hold the value of the returned id field from the subscription you want to use: `az account set --subscription="${SUBSCRIPTION_ID}"`
- Now you can create a service principal for use with Terraform using `az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/${subscriptionId}"`
- This command should return with key:value pairs appId,displayName,name,password & tenant .

```
{
 "appId": "xxxx-xxxx-xxxx-xxxx-xxxx-xxxx ",
 "displayName": "azure-cli-2019-11-18-09-29-52",
 "name": "http://azure-cli-2019-11-18-09-29-52",
 "password": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
 "tenant": "xxxxx-xxxx-xxxx-xxx-xxx-xxxx-xxxx"
}
```

- To configure Terraform to use our Azure AD service principal, set the following environment variables, which are then used by the Azure Terraform modules.
  ARM_SUBSCRIPTION_ID
  ARM_CLIENT_ID
  ARM_CLIENT_SECRET
  ARM_TENANT_ID



We are now ready to proceed using Terraform to build our Azure infrastructure.

Create Linux infrastructure for our Kubernetes Cluster

Terraform allows you to define and create complete infrastructure deployments in Azure.

Using Terraform, we will provision the following things:

- A resource group with one virtual network in it
- One subnet within the virtual network;
- One network security group.
- Three VMs; 1 is going to be Kubernetes master node and 2 worker nodes.
- Three network interfaces, one for each VM. Each interface will have a public IP.
- An Azure Load balancer for our Jenkins installation; containing back end pool, load balancer probe, load balancer rule and dedicated IP and A record for Jenkins.

At first, we will create variables file. In the **variables.tf** file, we declare all the variables that we will use in all our Terraform configurations.

```
variable "resource_group_name" {
  description = "Resource group name that will contain all
resources"
}

variable "location" {
  description = "The Azure region for the resource provisioning"
}

variable "vnet_cidr" {
  description = "CIDR block for Virtual Network"
}

variable "subnet_cidr" {
  description = "CIDR block for Subnet within a Virtual
Network"
}

variable "vm_username" {
  description = "Enter admin username to SSH into VM"
}

variable "ssh_key" {
  description = "Enter ssh key to access VM"
}

variable "jenkins_dnslabel" {
  description = "Jenkins dns applied FQDN"
}

variable "environment" {
  description = "Environment Tag"
}
```

We will assign values to variables by declaring them in another file: **terraform.tfvars**.

```
resource_group_name = "Project1-RG"
location = "West Europe"
vnet_cidr = "10.10.0.0/16"
subnet_cidr = "10.10.1.0/24"
jenkins_dnslabel = "albi-jenkins"
vm_username = "albi"
ssh_key = xxxxxx
environment = "Project1"
```

Azure connection and resource group

The provider section tells Terraform to use an Azure provider. Since this is a new project, we are going to create a new Resource Group.

```
provider "azurerm" {
}

resource "azurerm_resource_group" "Project1" {
    name     = var.resource_group_name
    location = var.location

    tags = {
        environment = var.environment
    }
}
```

Create virtual network and subnet

We will create a virtual network and a subnet where we will place our virtual machines.

```
resource "azurerm_virtual_network" "Project1-network" {
    name                = "myVnet"
    address_space       = [var.vnet_cidr]
    location            = var.location
    resource_group_name = azurerm_resource_group.Project1.name

    tags = {
        environment = var.environment
    }
}

resource "azurerm_subnet" "Project1-subnet" {
    name                 = "mySubnet"
    resource_group_name  = azurerm_resource_group.Project1.name
    virtual_network_name = azurerm_virtual_network.Project1-network.name
    address_prefix       = var.subnet_cidr
}
```

Create Network Security Group

Network Security Groups control the flow of network traffic in and out of your VM. The following section creates a network security group named myNetworkSecurityGroup and defines a rule to allow SSH traffic on TCP port 22 and Jenkins internal port on TCP port 30000.

```
resource "azurerm_network_security_group" "Project1-NSG" {
   name                = "myNetworkSecurityGroup"
   location            = var.location
   resource_group_name = azurerm_resource_group.Project1.name

   security_rule {
      name                       = "SSH"
      priority                   = 1001
      direction                  = "Inbound"
      access                     = "Allow"
      protocol                   = "Tcp"
      source_port_range          = "*"
      destination_port_range     = "22"
      source_address_prefix      = "*"
      destination_address_prefix = "*"
   }
   security_rule {
      name                       = "Jenkins"
      priority                   = 1011
      direction                  = "Inbound"
      access                     = "Allow"
      protocol                   = "Tcp"
      source_port_range          = "*"
      destination_port_range     = "30000"
      source_address_prefix      = "*"
      destination_address_prefix = "*"
   }

   tags = {
      environment = var.environment
   }
}
```

Create public IP address

To access our VMs from the internet, we are going to create Public IPs and assign it to them. We are also going to create another public IP for our load balancer which is going to be used for Jenkins. In our Jenkins Public IP section, we are going to add a domain_name_label so a FQDN will be created on Azure DNS servers.

```
resource "azurerm_public_ip" "masterkube-pubIP" {
   name                    = "masterkube-pubIP"
   location                = var.location
   resource_group_name         = azurerm_resource_group.Project1.name
   allocation_method           = "Static"

   tags = {
      environment = var.environment
   }
}

resource "azurerm_public_ip" "Node1-pubIP" {
   name                    = "Node1-pubIP"
   location                = var.location
   resource_group_name         = azurerm_resource_group.Project1.name
   allocation_method           = "Static"

   tags = {
      environment = var.environment
   }
}

resource "azurerm_public_ip" "Node2-pubIP" {
   name                    = "Node2-pubIP"
   location                = var.location
   resource_group_name         = azurerm_resource_group.Project1.name
   allocation_method           = "Static"

   tags = {
      environment = var.environment
   }
}

resource "azurerm_public_ip" "Jenkins-PublicIP" {
   name                    = "Jenkins-PublicIP"
   location                = var.location
   resource_group_name         = azurerm_resource_group.Project1.name
   allocation_method           = "Static"
   domain_name_label   = var.jenkins_dnslabel

  tags = {
      environment = var.environment
   }
}
```

Create Load Balancer for Jenkins

We are creating an Azure load balancer and rules to serve the Jenkins application and attach it to the public IP address configured earlier. Components for the load balancer are the back end pool, which contains network cards of our 3 VMs, a load balancer probe which will check health status of Jenkins internal port, and the load balancing rule which will gather all the information and expose Jenkins on its public ip and http port 80.

```
resource "azurerm_lb" "Jenkins-LB" {
  name                = "Jenkins-LB"
  location            = var.location
  resource_group_name = azurerm_resource_group.Project1.name
  frontend_ip_configuration {
    name                 = "Jenkins-PublicIP"
    public_ip_address_id = azurerm_public_ip.Jenkins-PublicIP.id
  }
   tags = {
       environment = var.environment
    }
}
resource "azurerm_lb_backend_address_pool" "backendpool" {
  resource_group_name = azurerm_resource_group.Project1.name
  loadbalancer_id     = azurerm_lb.Jenkins-LB.id
  name                = "BackEndAddressPool"
}
resource "azurerm_lb_probe" "health_probe" {
  resource_group_name = azurerm_resource_group.Project1.name
  loadbalancer_id     = azurerm_lb.Jenkins-LB.id
  name                = "health_probe"
  protocol            = "tcp"
  port                = 30000
  interval_in_seconds = 5
  number_of_probes    = 2
}
resource "azurerm_lb_rule" "lb_rule" {
  resource_group_name            = azurerm_resource_group.Project1.name
  loadbalancer_id                = azurerm_lb.Jenkins-LB.id
  name                           = "LBRule"
  protocol                       = "tcp"
  frontend_port                  = 80
  backend_port                   = 30000
  frontend_ip_configuration_name = "Jenkins-PublicIP"
  enable_floating_ip             = false
  backend_address_pool_id        = azurerm_lb_backend_address_pool.backendpool.id
  idle_timeout_in_minutes        = 5
  probe_id                       = azurerm_lb_probe.health_probe.id
  depends_on                     = [azurerm_lb_probe.health_probe]
}
```

Create virtual network interface card

A virtual network interface card (NIC) connects our VM to the given virtual network, public IP address, network security group and the load balancer we created. We are going to assign static IP address to each VM.

```
resource "azurerm_network_interface" "masterkube-nic" {
   name                = "masterkube-nic"
   location            = var.location
   resource_group_name      = azurerm_resource_group.Project1.name
   network_security_group_id = azurerm_network_security_group.Project1-NSG.id

   ip_configuration {
      name                       = "masterkube-nicConfiguration"
      subnet_id                  = azurerm_subnet.Project1-subnet.id
      private_ip_address_allocation = "Static"
                  private_ip_address = "10.10.1.10"
                  load_balancer_backend_address_pools_ids =
[azurerm_lb_backend_address_pool.backendpool.id]
      public_ip_address_id       = azurerm_public_ip.masterkube-pubIP.id
   }

   tags = {
      environment = var.environment
   }
}
resource "azurerm_network_interface" "node1-nic" {
   name                = "node1-nic"
   location            = var.location
   resource_group_name      = azurerm_resource_group.Project1.name
   network_security_group_id = azurerm_network_security_group.Project1-NSG.id

   ip_configuration {
      name                       = "node1-nicConfiguration"
      subnet_id                  = azurerm_subnet.Project1-subnet.id
      private_ip_address_allocation = "Static"
                  private_ip_address = "10.10.1.11"
                  load_balancer_backend_address_pools_ids =
[azurerm_lb_backend_address_pool.backendpool.id]
      public_ip_address_id       = azurerm_public_ip.Node1-pubIP.id
   }

   tags = {
      environment = var.environment
   }
}
```

```
resource "azurerm_network_interface" "node2-nic" {
    name                = "node2-nic"
    location            = var.location
    resource_group_name       = azurerm_resource_group.Project1.name
    network_security_group_id = azurerm_network_security_group.Project1-NSG.id

    ip_configuration {
        name                          = "node1-nicConfiguration"
        subnet_id                     = azurerm_subnet.Project1-subnet.id
        private_ip_address_allocation = "Static"
        private_ip_address = "10.10.1.12"
        load_balancer_backend_address_pools_ids = [azurerm_lb_backend_address_pool.backendpool.id]
        public_ip_address_id          = azurerm_public_ip.Node2-pubIP.id
    }

    tags = {
        environment = var.environment
    }
}
```

Create storage account for diagnostics

This following configuration will configure a dedicated storage account to store boot diagnostics for all our VMs. The storage account will have a random generated ID.

```
resource "random_id" "randomId" {
    keepers = {
        resource_group = azurerm_resource_group.Project1.name
    }

    byte_length = 8
}

resource "azurerm_storage_account" "Project1-StorageAcc" {
    name                        = "diag${random_id.randomId.hex}"
    resource_group_name         = azurerm_resource_group.Project1.name
    location                    = var.location
    account_tier                = "Standard"
    account_replication_type    = "LRS"

    tags = {
        environment = var.environment
    }
}
```

Create virtual machines

The final step is to create all VMs and use all the resources created. The following section creates an availability set, 3 VMs and attaches the virtual NICs dedicated for each VM. Centos 7.5 image is used, and a user named albi is created. Authentication to VMs will be through a dedicated SSH key.

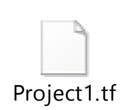
```
resource "azurerm_availability_set" "avset" {
            name                        = "avset"
            location                    = var.location
            resource_group_name         = azurerm_resource_group.Project1.name
            platform_fault_domain_count  = 3
            platform_update_domain_count = 3
            managed                      = true
}
```

```
resource "azurerm_virtual_machine" "MasterKube" {
  name                  = "MasterKube"
  location              = var.location
  resource_group_name   = azurerm_resource_group.Project1.name
  network_interface_ids = [azurerm_network_interface.masterkube-nic.id]
  vm_size               = "Standard_B2s"
        availability_set_id   = azurerm_availability_set.avset.id

  storage_os_disk {
    name              = "MasterKubeOsDisk"
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "Premium_LRS"
  }

  storage_image_reference {
    publisher = "OpenLogic"
    offer     = "CentOS"
    sku       = "7.5"
    version   = "latest"
  }

  os_profile {
    computer_name  = "MasterKube"
    admin_username = var.vm_username
  }

  os_profile_linux_config {
    disable_password_authentication = true
    ssh_keys {
      path     = "/home/albi/.ssh/authorized_keys"
      key_data = var.ssh_key
    }
  }

  boot_diagnostics {
    enabled = "true"
    storage_uri = azurerm_storage_account.Project1-StorageAcc.primary_blob_endpoint
  }

  tags = {
    environment = var.environment
  }
}
```

```
resource "azurerm_virtual_machine" "Node1" {
  name                  = "NODE1"
  location              = var.location
  resource_group_name   = azurerm_resource_group.Project1.name
  network_interface_ids = [azurerm_network_interface.node1-nic.id]
  vm_size               = "Standard_B2s"
        availability_set_id   = azurerm_availability_set.avset.id

  storage_os_disk {
    name              = "Node1OsDisk"
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "Premium_LRS"
  }

  storage_image_reference {
    publisher = "OpenLogic"
    offer     = "CentOS"
    sku       = "7.5"
    version   = "latest"
  }

  os_profile {
    computer_name  = "NODE1"
    admin_username = var.vm_username
  }

  os_profile_linux_config {
    disable_password_authentication = true
    ssh_keys {
      path     = "/home/albi/.ssh/authorized_keys"
      key_data = var.ssh_key
    }
  }

  boot_diagnostics {
    enabled = "true"
    storage_uri = azurerm_storage_account.Project1-StorageAcc.primary_blob_endpoint
  }

  tags = {
    environment = var.environment
  }
}
```

```
resource "azurerm_virtual_machine" "Node2" {
  name                  = "NODE2"
  location              = var.location
  resource_group_name   = azurerm_resource_group.Project1.name
  network_interface_ids = [azurerm_network_interface.node2-nic.id]
  vm_size               = "Standard_B2s"
        availability_set_id   = azurerm_availability_set.avset.id

  storage_os_disk {
    name              = "Node2OsDisk"
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "Premium_LRS"
  }

  storage_image_reference {
    publisher = "OpenLogic"
    offer     = "CentOS"
    sku       = "7.5"
    version   = "latest"
  }

  os_profile {
    computer_name  = "NODE2"
    admin_username = var.vm_username
  }

  os_profile_linux_config {
    disable_password_authentication = true
    ssh_keys {
      path     = "/home/albi/.ssh/authorized_keys"
      key_data = var.ssh_key
    }
  }

  boot_diagnostics {
    enabled = "true"
    storage_uri = azurerm_storage_account.Project1-StorageAcc.primary_blob_endpoint
  }

  tags = {
    environment = var.environment
  }
}
```

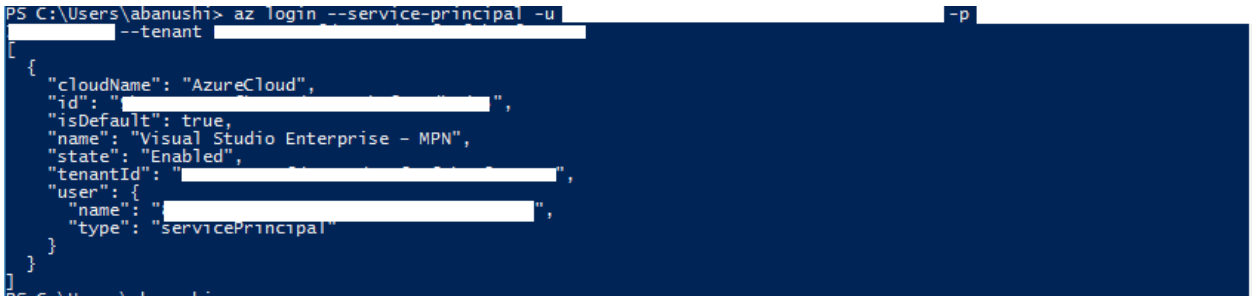
To bring all these sections together and see Terraform in action, create a file called **Project1.tf** and put every configuration on this file.



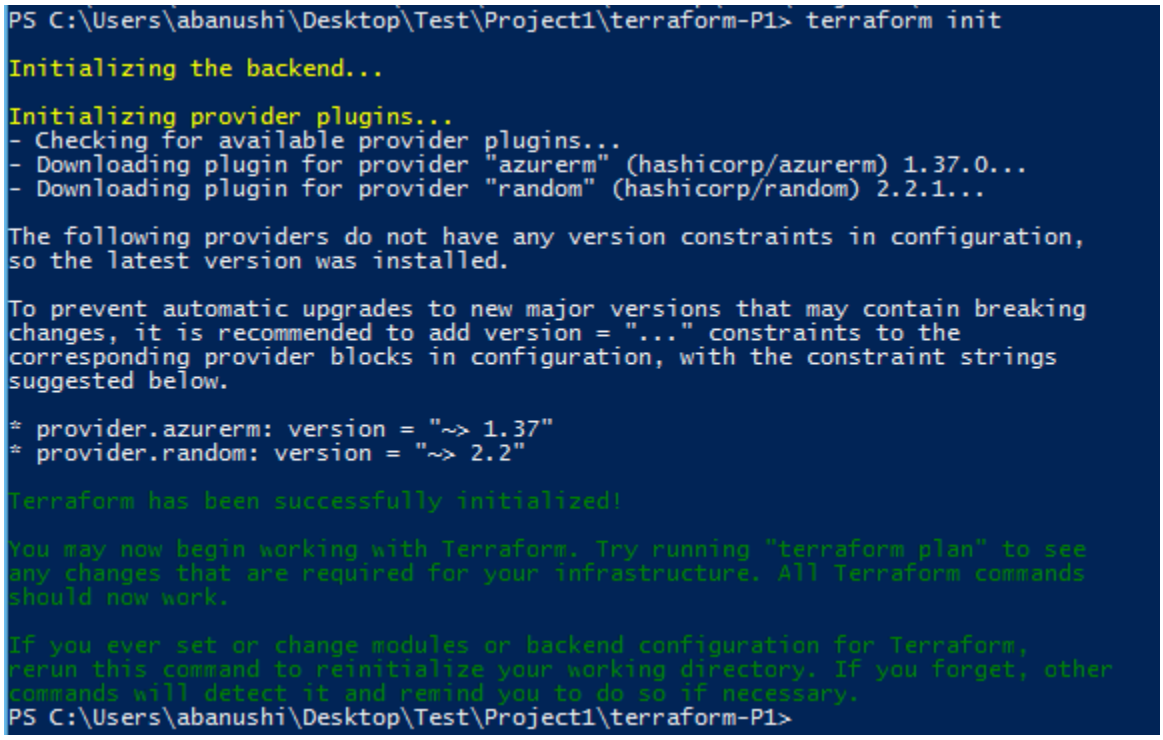
Build and deploy the infrastructure

With all the components of Terraform infrastructure created, we put all our files in a single directory and start deploying the configurations.

- Open a powershell terminal and login with the service principal we previously created:



- Go to the directory where we have placed our Terraform files and initialize terraform.

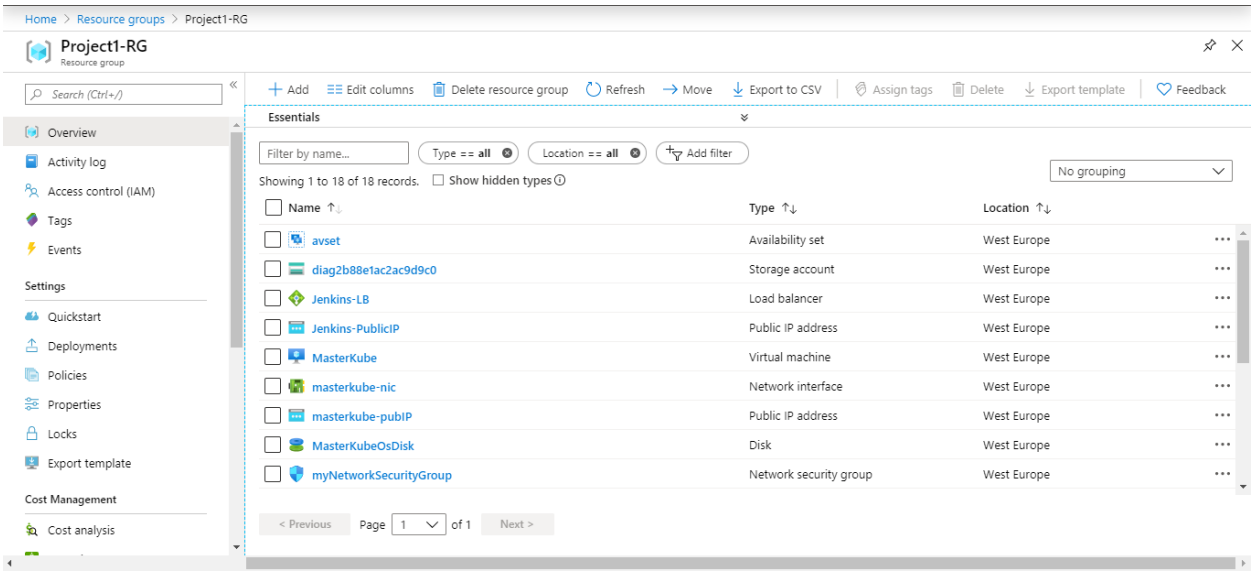


- The next step is to have Terraform review and validate the template. This step compares the requested resources to the state information saved by Terraform and then outputs the planned execution: terraform plan -out newplan

```
Plan: 21 to add, 0 to change, 0 to destroy.

------------------------------------------------------------------------

This plan was saved to: newplan

To perform exactly these actions, run the following command to apply:
    terraform apply "newplan"

PS C:\Users\abanushi\Desktop\Test\Project1\terraform-P1>
```

- Now deploy infrastructure to Azure: terraform apply "newplan"

```
Apply complete! Resources: 21 added, 0 changed, 0 destroyed.

The state of your infrastructure has been saved to the path
below. This state is required to modify and destroy your
infrastructure, so keep it safe. To inspect the complete state
use the `terraform show` command.

State path: terraform.tfstate

Outputs:

Jenkins-FQDN = albi-jenkins.westeurope.cloudapp.azure.com
MasterKube-IP = 51.137.56.118
Node1-IP = 52.148.214.2
Node2-IP = 51.137.56.34
PS C:\Users\abanushi\Desktop\Test\Project1\terraform-P1>
```



Creating a Kubernetes Cluster

Kubernetes is a container orchestration system that manages containers at scale. Initially developed by Google based on its experience running containers in production, Kubernetes is open source and actively developed by a community around the world. Kubeadm automates the installation and configuration of Kubernetes components such as the API server, Controller Manager, and Kube DNS.

This cluster will include the following compute resources:

- One master node

The master node is responsible for managing the state of the cluster. It runs Etcd, which stores cluster data among components that schedule workloads to worker nodes.

- Two worker nodes

Worker nodes are the servers where your workloads (containerized applications and services) will run. A worker will continue to run your workload once they're assigned to it, even if the master goes down once scheduling is complete. A cluster's capacity can be increased by adding workers.

Installing Docker

On each of our already deployed virtual machines we are going to install Docker as our container runtime for the Kubernetes cluster. Docker version 18.06.2 is the latest validated version for Kubernetes. Docker will be installed using the follow commands on each of our servers.

```
# Install required packages.
sudo yum install yum-utils device-mapper-persistent-data lvm2

# Add Docker repository.
sudo yum-config-manager \
  --add-repo \
  https://download.docker.com/linux/centos/docker-ce.repo

#Update system and Install Docker CE.
sudo yum update && sudo yum install docker-ce-18.06.2.ce

## Create /etc/docker directory.
sudo mkdir /etc/docker
```

```
# Setup daemon. Run as root.
cat > /etc/docker/daemon.json <<EOF
{
 "exec-opts": ["native.cgroupdriver=systemd"],
 "log-driver": "json-file",
 "log-opts": {
  "max-size": "100m"
 },
 "storage-driver": "overlay2",
 "storage-opts": [
  "overlay2.override_kernel_check=true"
 ]
}
EOF

sudo mkdir -p /etc/systemd/system/docker.service.d

# Restart Docker
sudo systemctl daemon-reload
sudo systemctl restart docker

#Enable docker on startup
sudo systemctl enable --now docker
```

```
Installed:
  docker-ce.x86_64 0:18.06.2.ce-3.el7

Dependency Installed:
  audit-libs-python.x86_64 0:2.8.5-4.el7
  checkpolicy.x86_64 0:2.5-8.el7
  container-selinux.noarch 2:2.107-3.el7
  libcgroup.x86_64 0:0.41-21.el7
  libsemanage-python.x86_64 0:2.5-14.el7
  libtool-ltdl.x86_64 0:2.4.2-22.el7_3
  policycoreutils-python.x86_64 0:2.5-33.el7
  python-IPy.noarch 0:0.75-6.el7
  setools-libs.x86_64 0:3.3.8-4.el7
```

```
[albi@MasterKube ~]$ systemctl status docker.service
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor preset: disabled)
   Active: active (running) since Thu 2019-11-28 13:48:35 UTC; 1min 41s ago
     Docs: https://docs.docker.com
 Main PID: 117155 (dockerd)
   CGroup: /system.slice/docker.service
           ├─117155 /usr/bin/dockerd
           └─117162 docker-containerd --config /var/run/docker/containerd/containerd.toml
[albi@MasterKube ~]$
```

Install kubelet, kubeadm, and kubectl

The kubelet is the node agent that will run all the pods for us, including the kube-system pods. The kubeadm is a tool for deploying multi-node kubernetes clusters. And then the kubectl is the command line tool for interacting with Kubernetes. Install kubelet, kubeadm and kubectl on all 3 VMs.

```
#Add the kubernetes repo needed to find the kubelet, kubeadm and kubectl packages
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF

# Set SELinux in permissive mode. This disables SELinux since it is not fully supported by Kubernetes.
##Login as root
setenforce 0
sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/selinux/config

#Install kubelet, kubeadm, kubectl
sudo yum install -y kubelet kubeadm kubectl --disableexcludes=kubernetes

#Enable kubelet startup
sudo systemctl enable --now kubelet

#Set bridge IP tables value to 1. This will allow Kubernetes to set iptables rules for receiving bridged IPv4
and IPv6 network traffic on the nodes.
##Login as root.
cat <<EOF >  /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF

sudo sysctl --system
```

```
Installed:
  kubeadm.x86_64 0:1.16.3-0 kubectl.x86_64 0:1.16.3-0 kubelet.x86_64 0:1.16.3-0

Dependency Installed:
  conntrack-tools.x86_64 0:1.4.4-5.el7_7.2
  cri-tools.x86_64 0:1.13.0-0
  kubernetes-cni.x86_64 0:0.7.5-0
  libnetfilter_cthelper.x86_64 0:1.0.0-10.el7_7.1
  libnetfilter_cttimeout.x86_64 0:1.0.0-6.el7_7.1
  libnetfilter_queue.x86_64 0:1.0.2-2.el7_2
  socat.x86_64 0:1.7.3.2-2.el7

Complete!
[albi@NODE1 ~]$
```

Set up dhe Master Node

The follow commands will be applied in the master node only.

Initialize the cluster using the following command:

```
sudo kubeadm init --pod-network-cidr=10.244.0.0/16
```

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 10.10.1.10:6443 --token megdid.3z6nii9qg0y5y3z4 \
    --discovery-token-ca-cert-hash sha256:e2bb49271da9ff5173ae28e534acbf6da005b2eca0faae1816d5
51c7cd3ef877
[albi@MasterKube ~]$
```

Save the " kubeadm join" output command because we are going to use it to join worker nodes to the cluster.

Follow the instructions and create a kube directory:

```
mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Install pod network plugin (Flannel)

Each pod has its own IP address, and a pod on one node should be able to access a pod on another node using the pod's IP. Containers on a single node can communicate easily through a local interface. Communication between pods is more complicated, however, and requires a separate networking component that can transparently route traffic from a pod on one node to a pod on another.

This functionality is provided by pod network plugins. For this cluster, we will use **Flannel**.

Install Flannel on master node with the following command:

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```

```
[albi@MasterKube ~]$ kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/
Documentation/kube-flannel.yml
podsecuritypolicy.policy/psp.flannel.unprivileged created
clusterrole.rbac.authorization.k8s.io/flannel created
clusterrolebinding.rbac.authorization.k8s.io/flannel created
serviceaccount/flannel created
configmap/kube-flannel-cfg created
daemonset.apps/kube-flannel-ds-amd64 created
daemonset.apps/kube-flannel-ds-arm64 created
daemonset.apps/kube-flannel-ds-arm created
daemonset.apps/kube-flannel-ds-ppc64le created
daemonset.apps/kube-flannel-ds-s390x created
[albi@MasterKube ~]$
```

To pass bridged IPv4 traffic to iptables' chains. This is a requirement for some CNI plugins to work

```
sudo sysctl net.bridge.bridge-nf-call-iptables=1
```

Join worker nodes to the cluster

After initializing the master node and installing the pod network plugin worker nodes are ready to join the cluster. Join the worker nodes to the cluster using the "kubeadm join" command as sudo.

```
kubeadm join 10.10.1.10:6443 --token megdid.3z6nii9qg0y5y3z4 \
    --discovery-token-ca-cert-hash sha256:e2bb49271da9ff5173ae28e534acbf6da005b2eca0faae1816d551c7cd3ef877
```

We can check worker nodes status from the master node:

```
[albi@MasterKube ~]$ kubectl get nodes -o wide
NAME         STATUS   ROLES    AGE   VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE                KERNEL-VERSION                CONTAINER-RUNTIME
masterkube   Ready    master   15m   v1.16.3   10.10.1.10    <none>        CentOS Linux 7 (Core)   3.10.0-862.11.6.el7.x86_64    docker://18.6.2
node1        Ready    <none>   78s   v1.16.3   10.10.1.11    <none>        CentOS Linux 7 (Core)   3.10.0-862.11.6.el7.x86_64    docker://18.6.2
node2        Ready    <none>   41s   v1.16.3   10.10.1.12    <none>        CentOS Linux 7 (Core)   3.10.0-862.11.6.el7.x86_64    docker://18.6.2
[albi@MasterKube ~]$
```

The cluster is successfully set up.

Pods status: kubectl get pods --all-namespaces -o wide

```
[albi@MasterKube ~]$ kubectl get pods --all-namespaces -o wide
NAMESPACE     NAME                                 READY   STATUS    RESTARTS   AGE     IP           NODE
kube-system   coredns-5644d7b6d9-6gtdj             1/1     Running   0          19m     10.244.0.2   masterkube
kube-system   coredns-5644d7b6d9-wbzzj             1/1     Running   0          19m     10.244.0.3   masterkube
kube-system   etcd-masterkube                      1/1     Running   0          18m     10.10.1.10   masterkube
kube-system   kube-apiserver-masterkube            1/1     Running   0          18m     10.10.1.10   masterkube
kube-system   kube-controller-manager-masterkube   1/1     Running   0          18m     10.10.1.10   masterkube
kube-system   kube-flannel-ds-amd64-jppbt          1/1     Running   1          4m57s   10.10.1.11   node1
kube-system   kube-flannel-ds-amd64-rz7vt          1/1     Running   0          12m     10.10.1.10   masterkube
kube-system   kube-flannel-ds-amd64-srblh          1/1     Running   0          4m20s   10.10.1.12   node2
kube-system   kube-proxy-lgb7j                     1/1     Running   0          4m57s   10.10.1.11   node1
kube-system   kube-proxy-mvncp                     1/1     Running   0          19m     10.10.1.10   masterkube
kube-system   kube-proxy-wlm6x                     1/1     Running   0          4m20s   10.10.1.12   node2
kube-system   kube-scheduler-masterkube            1/1     Running   0          18m     10.10.1.10   masterkube
[albi@MasterKube ~]$
```

Install Jenkins inside Kubernetes

Jenkins is an open-source continuous integration and continuous delivery tool, which can be used to automate building, testing, and deploying software. It is widely considered the most popular automation server, being used by more than a million users worldwide.

We want to start by deploying a Jenkins master instance onto a Kubernetes cluster. We will use Jenkins' kubernetes plugin to scale Jenkins on the cluster by provisioning dynamic agents to accommodate its current workloads. The plugin will create a Kubernetes Pod for each build by launching an agent based on a specific Docker image. When the build completes, Jenkins will remove the Pod to save resources. Agents will be launched using JNLP (Java Network Launch Protocol), so we the containers will be able to automatically connect to the Jenkins master once up and running.

Building custom docker images for Jenkins

We will build custom docker images and push them to Docker Hub.

Create a file called Dockerfile-jenkins-master. Inside include the following build instructions. These instructions use the main Jenkins Docker image as a base and configure the plugins we will use to deploy onto a Kubernetes cluster:

```
FROM jenkins/jenkins:lts

# Plugin for scaling Jenkins agents
RUN /usr/local/bin/install-plugins.sh kubernetes

USER jenkins
```

Save and close the file.

Create a new Dockerfile for the first Jenkins agent called Dockerfile-jenkins-slave-jnlp1

```
FROM jenkins/jnlp-slave

ENTRYPOINT ["jenkins-slave"]
```

Save and close the file.

Create a new Dockerfile for the second Jenkins agent called Dockerfile-jenkins-slave-jnlp2

```
FROM jenkins/jnlp-slave

ENTRYPOINT ["jenkins-slave"]
```

Save and close the file.

Now we are ready to build the images and push to docker hub:

```
docker build -f Dockerfile-jenkins-master -t albialb/jenkins-master .
```

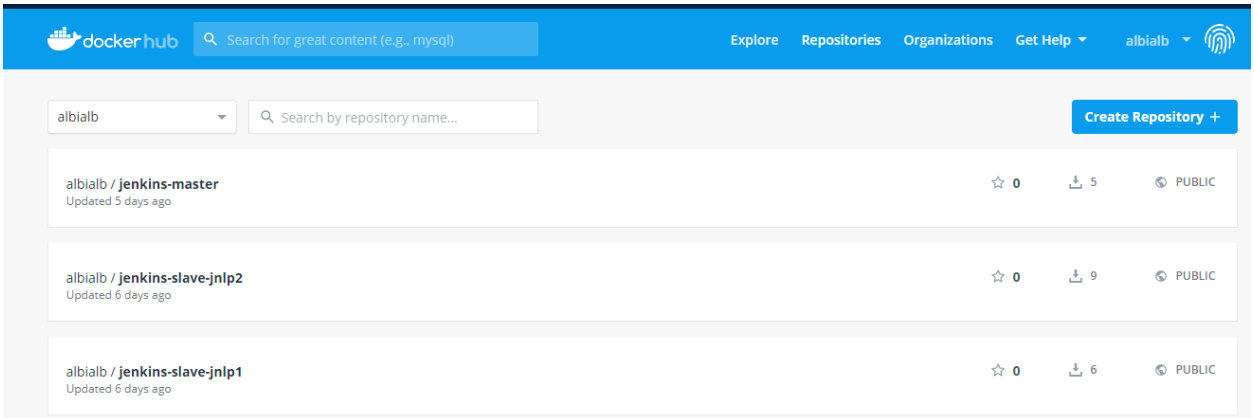
Login to docker hub using account: docker login
Push images to docker hub:

```
docker push albialb/jenkins-master
```

Build and push the agents images:

```
docker build -f Dockerfile-jenkins-slave-jnlp1 -t albialb/jenkins-slave-jnlp1 .
docker push albialb/jenkins-slave-jnlp1
```

```
docker build -f Dockerfile-jenkins-slave-jnlp2 -t albialb/jenkins-slave-jnlp2 .
docker push albialb/jenkins-slave-jnlp2
```



Deploying Jenkins to the cluster

Start by creating a file to define the Jenkins deployment:

vi jenkins-deployment.yaml

and paste the following:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jenkins
spec:
  replicas: 1
  selector:
   matchLabels:
    app: jenkins
  template:
    metadata:
      labels:
        app: jenkins
    spec:
      serviceAccountName: jenkins-robot
      containers:
        - name: jenkins
          image: albialb/jenkins-master
          env:
            - name: JAVA_OPTS
              value: -Djenkins.install.runSetupWizard=false
          ports:
            - name: http-port
              containerPort: 8080
            - name: jnlp-port
              containerPort: 50000
          volumeMounts:
            - name: jenkins-home
              mountPath: /var/jenkins_home
      volumes:
        - name: jenkins-home
          emptyDir: {}
```

or download using the following:

wget https://raw.githubusercontent.com/albi-github/project01/master/jenkins-deployment.yaml

Next, create a file to configure the two services we will create.

vi jenkins-services.yaml

and paste the following:

```
apiVersion: v1
kind: Service
metadata:
  name: jenkins
spec:
  type: NodePort
  ports:
    - nodePort: 30000
      port: 80
      targetPort: 8080
  selector:
    app: jenkins

---

apiVersion: v1
kind: Service
metadata:
  name: jenkins-jnlp
spec:
  type: ClusterIP
  ports:
    - port: 50000
      targetPort: 50000
  selector:
    app: jenkins
```

This allows Jenkins service to be accessible from outside each Kubernetes node on port 30000.

or download using the following:

wget https://raw.githubusercontent.com/albi-github/project01/master/jenkins-services.yaml

Jenkins provides two services that the cluster needs access to. Deploy these services separately so they can be individually managed and named.

- An externally-exposed NodePort service on port 30000 external users to access the Jenkins user interface. This type of service can be load balanced by an HTTP load balancer.

- An internal, private ClusterIP service on port 50000 that the Jenkins executors use to communicate with the Jenkins master from inside the cluster.

Next, create a service account for Jenkins using the following yaml.

```
apiVersion: v1
kind: ServiceAccount
metadata:
 name: jenkins-robot
```

Give cluster admin rights to this service account:

kubectl create clusterrolebinding add-on-cluster-admin \

--clusterrole=cluster-admin \

--serviceaccount=default:jenkins-robot

```
[albi@MasterKube ~]$ kubectl create -f svc-account.yaml
serviceaccount/jenkins-robot created
[albi@MasterKube ~]$  kubectl create clusterrolebinding add-on-cluster-admin \
>    --clusterrole=cluster-admin \
>    --serviceaccount=default:jenkins-robot
clusterrolebinding.rbac.authorization.k8s.io/add-on-cluster-admin created
```

We now have the following files:

```
[albi@MasterKube ~]$ ls
jenkins-deployment.yaml  jenkins-services.yaml  svc-account.yaml
[albi@MasterKube ~]$
```

Deploy both Jenkins application and Jenkins services using the following command:

kubectl apply -f Jenkins-deployment.yaml

kubectl apply -f Jenkins-services.yaml

```
[albi@MasterKube ~]$ kubectl apply -f jenkins-deployment.yaml
deployment.apps/jenkins created
[albi@MasterKube ~]$ kubectl apply -f jenkins-services.yaml
service/jenkins created
service/jenkins-jnlp created
[albi@MasterKube ~]$
```

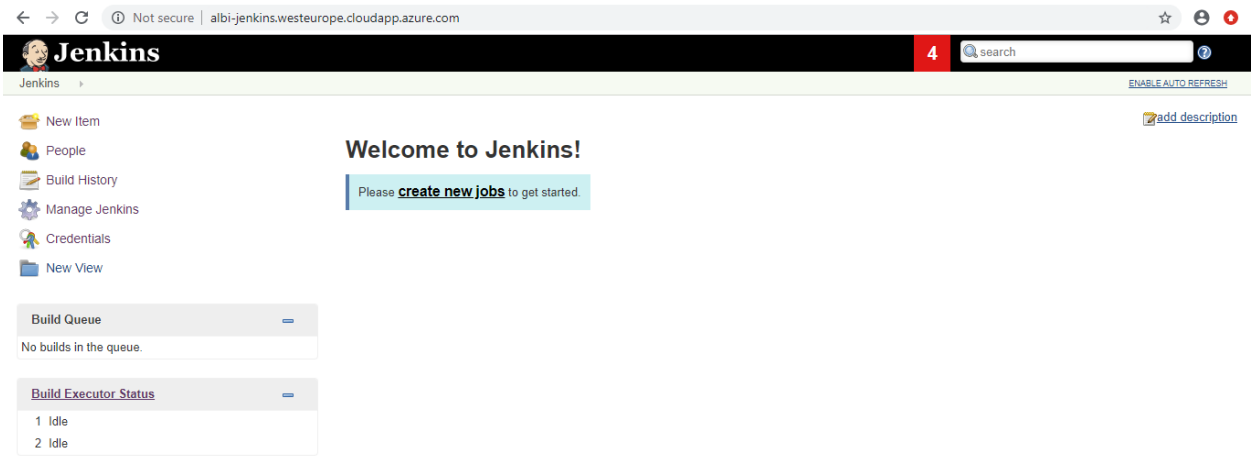
Status:

```
[albi@MasterKube ~]$ kubectl get pods --all-namespaces -o wide
NAMESPACE     NAME                              READY   STATUS    RESTARTS   AGE     IP           NODE
default       jenkins-778dfbb7cf-h9gc4          1/1     Running   0          3m49s   10.244.2.2   node2
```

```
[albi@MasterKube ~]$ kubectl get services --all-namespaces -o wide
NAMESPACE     NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                  AGE    SELECTOR
default       jenkins      NodePort    10.110.11.194   <none>        80:30000/TCP             4m9s   app=jenkins
default       jenkins-jnlp ClusterIP   10.111.97.111   <none>        50000/TCP                4m9s   app=jenkins
default       kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP                  112m   <none>
kube-system   kube-dns     ClusterIP   10.96.0.10      <none>        53/UDP,53/TCP,9153/TCP   112m   k8s-app=kube-dns
[albi@MasterKube ~]$
```

We can now open Jenkins web interface using the FQDN of the Load Balancer we created earlier using Terraform:

albi-jenkins.westeurope.cloudapp.azure.com

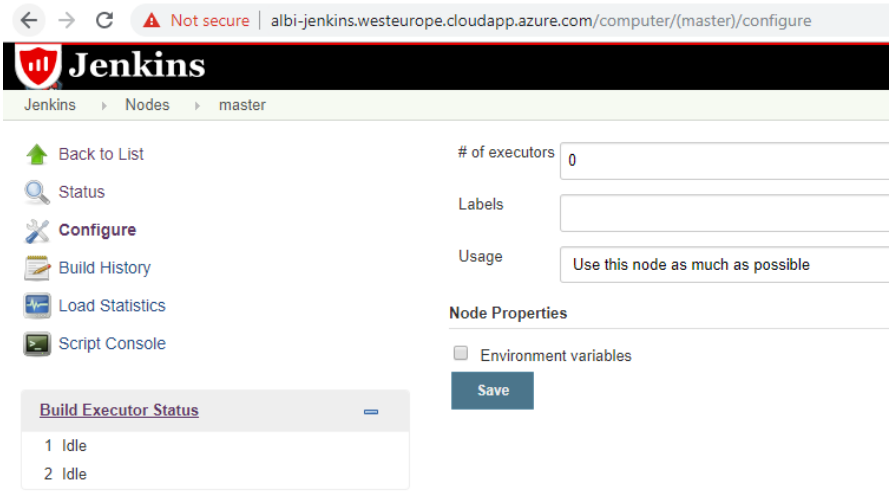


Configuring Jenkins

With the Jenkins master up and running, we can go ahead and configure dynamic build agents to automatically spin up Pods as necessary.
In the Jenkins UI, under Build Executor Status on the left side, two executors are configured by default, waiting to pick up build jobs. These are provided by the Jenkins master.

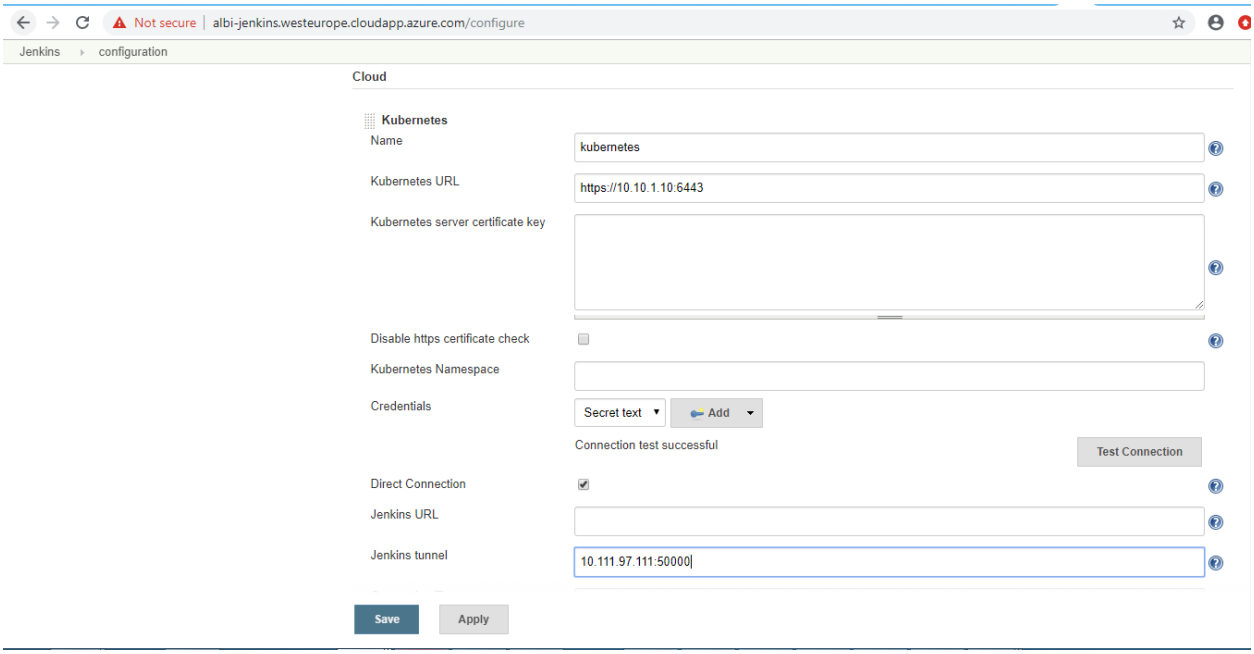
The master instance should only be in charge of scheduling build jobs, distributing the jobs to agents for execution, monitoring the agents, and getting the build results. Since we don't want our master instance to execute builds, we will disable these.

- Click on Manage Jenkins than Manage Nodes.
- Edit Jenkins Master, set # of executors to 0 and click Save.

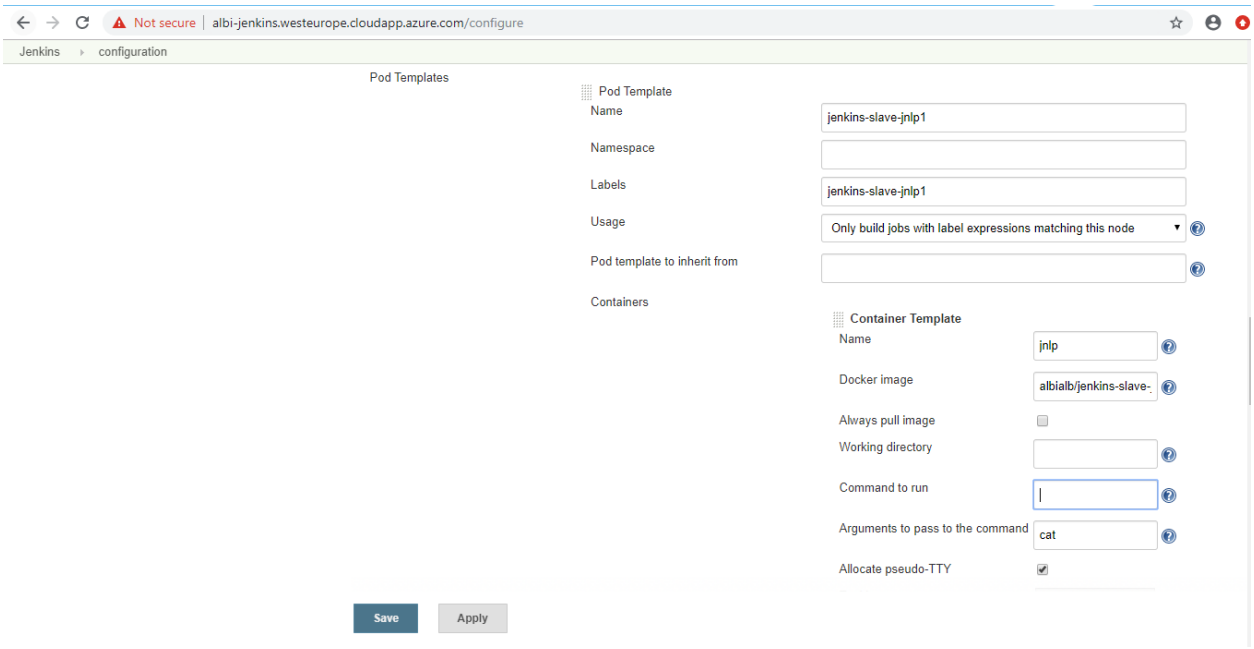


Configuring the Jenkins Kubernetes Plugin

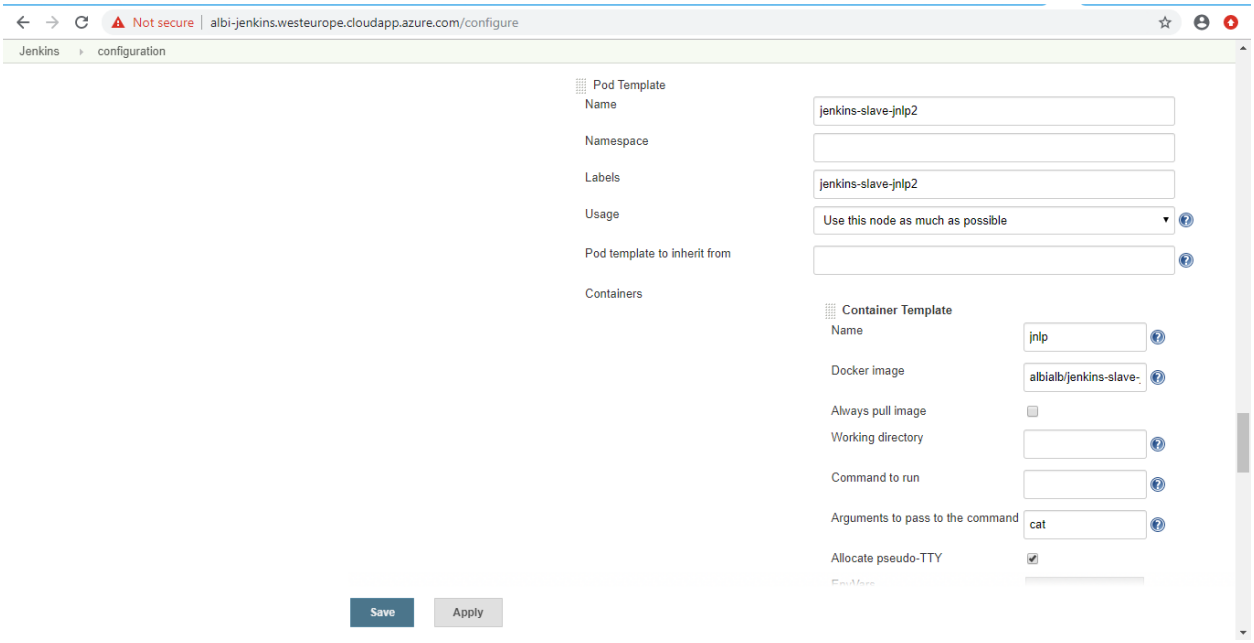
- In the main Jenkins dashboard, click on Manage Jenkins, followed by Manage Plugins:
- Go to Manage Jenkins and select Configure System:
- In the Cloud section at the bottom of the page. Click on Add a new cloud and select Kubernetes.
- On the form that follows, in the Kubernetes URL field, enter https:// followed by the cluster endpoint IP address.
- Under Credentials, click the Add button and select Kubernetes Service Account.
- Next, in the Jenkins tunnel field, enter the IP address and port of Jenkins JNLP service.



- Click the Add Pod Template button, and select Kubernetes Pod Template.
- Fill out the Name and Labels fields with unique values to identify your first agent.
- Click the Add Container button and select Container Template. In the section that appears, fill out the following fields

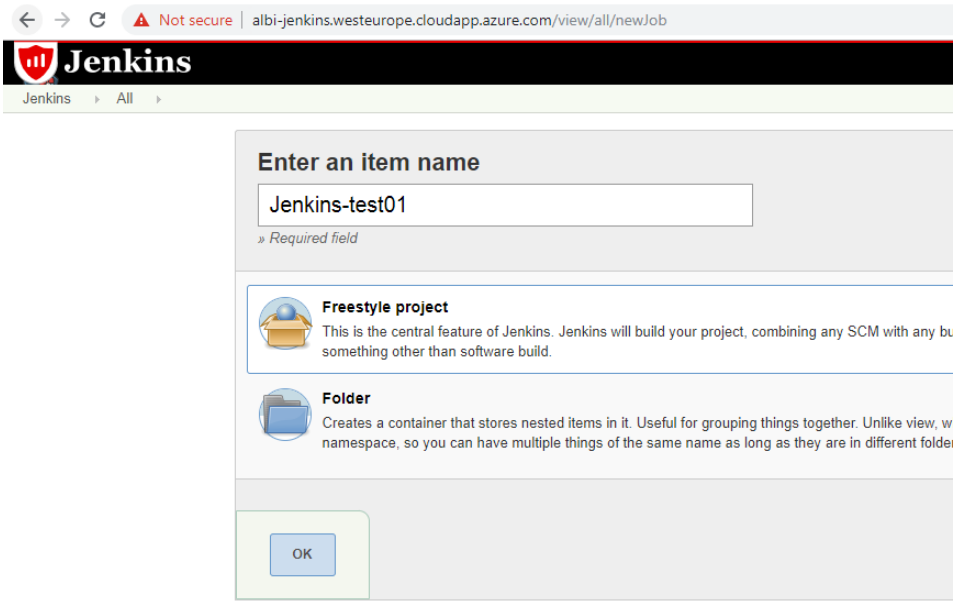


- Click again Add Pod Template, select Kubernetes Pod Template. Fill again for the second slave.

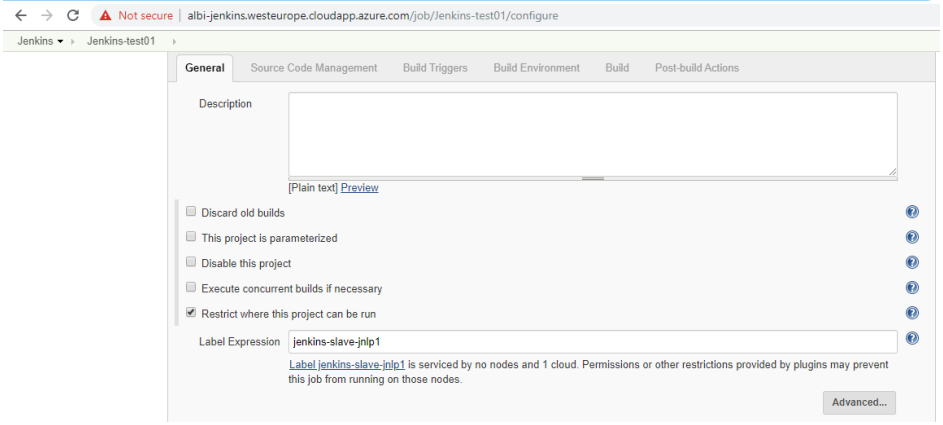


Testing Jenkins build jobs

On the main Jenkins page, click New Item on the left side. Enter a name for the first build of your first agent. Select Freestyle project and click the OK button



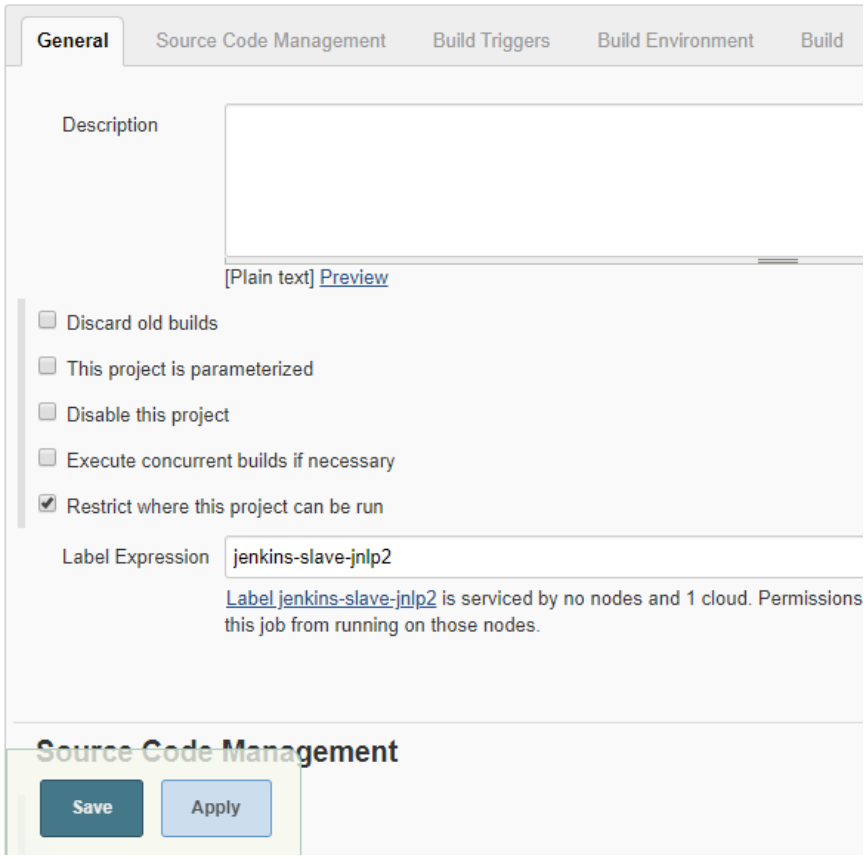
On the next page, in the Label Expression field, type the label of first Jenkins agent image.

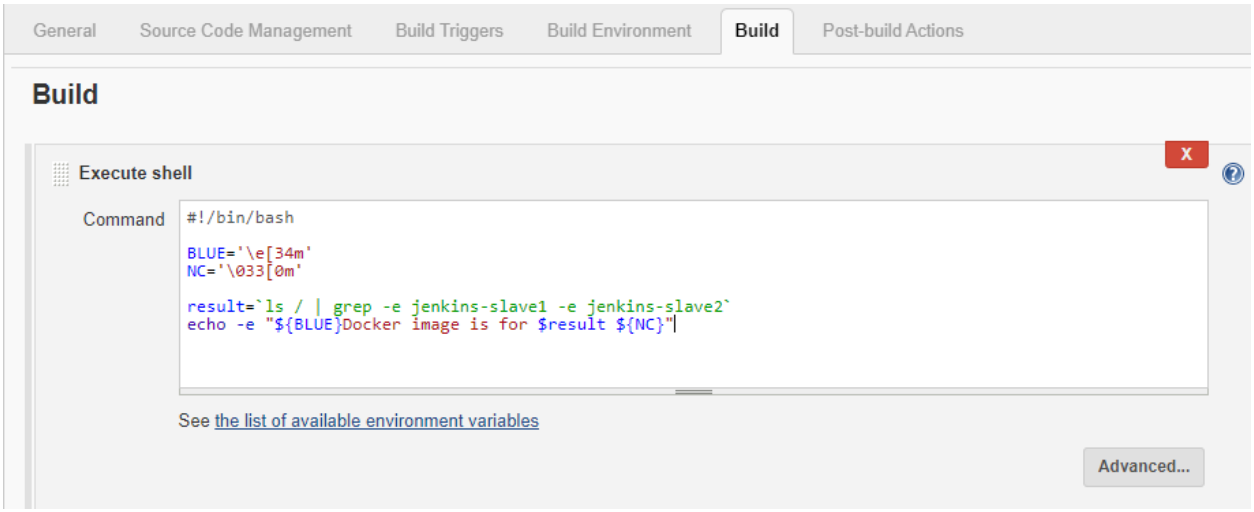


In the Build section, click Add build step and select Execute shell. Paste a demo script in the text box that appears:

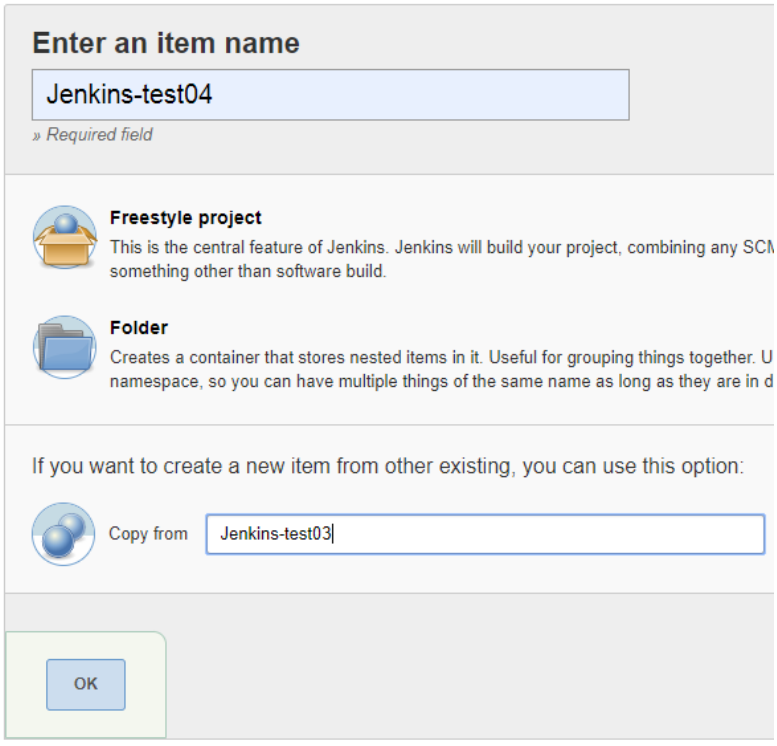
- Create another job for the first agent by clicking New Item, filling out a new name, and using the Copy from field to copy from your first build.

- Configure the first job for your second Jenkins agent. Click New Item, select a name for the first job for the second agent, and copy the job from your first agent again. This time, we will modify the fields on the configuration page before saving.

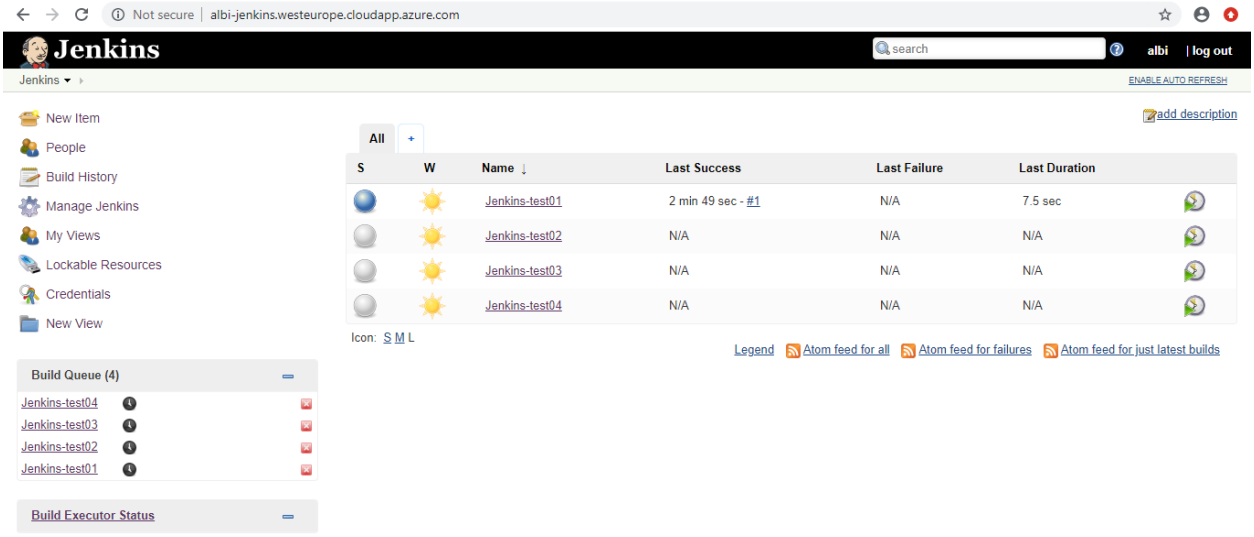




- Create another job for the second agent by clicking New Item, filling out a new name, and using the Copy from field to copy from your first build.



Run all created jobs:




```
[albi@MasterKube ~]$ kubectl get pods --all-namespaces -o wide
NAMESPACE     NAME                                READY   STATUS        RESTARTS   AGE   IP            NODE
default       jenkins-79f7c894b4-9rpvw            1/1     Running       0          21m   10.244.2.24   node2
default       jenkins-slave-jnlp1-gwzfw           0/1     Terminating   0          15s   10.244.2.26   node2
default       jenkins-slave-jnlp1-t9dd2           0/1     Terminating   0          15s   10.244.1.32   node1
default       jenkins-slave-jnlp2-5s67j           1/1     Running       0          5s    10.244.1.33   node1
default       jenkins-slave-jnlp2-ktk1w           1/1     Running       0          5s    10.244.2.27   node2
```

Build status Success.