# Practice M5: HashiCorp Tools

During this practice we will assume that we are working in Linux environment. It could be a physical machine or a virtual one. The distribution of choice is not that important, but it will be better to stick to some of the well supported distributions.

Most of the steps can be executed in Windows and/or macOS environment as well either directly or in a VM.

## Part 1: Packer and Vault

#### Install Packer

There are many ways to install **Packer** under Linux, but for the purpose of our exercise we will use:

* Go to <http://packer.io/downloads.html>, check which is the latest version, and copy the link
* Open a terminal window / session and execute:

**wget** [**https://releases.hashicorp.com/packer/1.3.3/packer\_1.3.3\_linux\_amd64.zip**](https://releases.hashicorp.com/packer/1.3.3/packer_1.3.3_linux_amd64.zip)

* Then extract the zip archive

**unzip packer\_1.3.3\_linux\_amd64.zip**

* Move the file to a location, part of your **PATH** variable, or include the target folder in your **PATH**

**sudo mv packer /usr/local/bin/**

* Now check that everything is okay, by executing:

**packer**

#### Packer and Docker

Let’s create two **Docker** images with the help of **Packer**. First, we will barely modify the original ubuntu image:

* Examine the file **ubuntu-packer.json**
* Now, let’s check if the file is valid

**packer validate ubuntu-packer.json**

* We can also inspect the file

**packer inspect ubntu-packer.json**

* Let’s build the new image

**packer build ubuntu-packer.json**

* Now, we can check it there are any containers left

**docker container ps -a**

* There should not be any images left, as the image is created, committed, and then removed. Now we will check if we have the new image correctly tagged:

**docker image ps**

* If we spin up a container out of it with the following command:

**docker run -it --rm --name dof-test ubuntu-packer:0.1 bash**

* And check for our changes:

**cat /root/readme.txt**

* Finally, exit the container

**exit**

Okay, now we will create an extended image and we will push the result to the Docker Hub:

* Examine the **Dockerfile.sample** file and compare its content to the **nginx-packer.json** file
* We can try and build image based on the **Dockerfile**

**mv Dockerfile.sample Dockerfile**

**docker image build -t ubuntu-nginx:docker .**

**docker container run -d --name dof-web1 -p 8001:80 ubuntu-nginx:docker**

* Now open a browser tab and navigate to <http://localhost:8001>
* We can use Packer and build alternative image:

**packer build -var 'vtag=packer' nginx-packer.json**

* Then spin up a container:

**docker container run -d --name dof-web2 -p 8002:80 ubuntu-nginx:packer**

* Now open a browser tab and navigate to <http://localhost:8002>
* Both tabs should have the same look and feel
* We can clean up everything, assuming that we do not need any containers, with:

**docker container rm --force $(docker container ps -aq)**

Currently our artefact stays in our local Docker repository, we can extend the pipeline to publish the image to Docker Hub:

* Examine the **nginx-packer-push.hub** file
* Execute:

**packer build -var 'vrepository=shekeriev/nginx-packer' -var 'vtag=latest' nginx-packer.json**

* If we substitute the repository value with one, corresponding to our Docker Hub repository, and we visit <https://hub.docker.com>, we shall see the image

#### Packer and AWS (+ Terraform)

Most probably in the real life, we would end up in a need to build and publish custom image to a cloud provider. Let’s experiment with a very simple setup against AWS EC2:

* Examine the file
* Then create an AWS user with **AmazonEC2FullAccess** permissions and write down the access and security keys
* Export the keys as variables – **AWS\_ACCESS\_KEY\_ID** and **AWS\_SECRET\_ACCESS\_KEY**
* Initiate the build process:

**packer build aws-packer.json**

* Go to the AWS console and browse for the new image

Now that we have our own image, let’s try to use it in a simple Terraform configuration:

* Examine the **main.tf** file
* Execute consequently the following commands:

**terraform init**

**terraform plan**

**terraform apply**

* Open a browser tab and navigate to the Public IP or Public DNS
* Now, clean up the infrastructure:

**terraform destroy**

* And the AMI, which we can do in the web console or on the command line:

**aws ec2 deregister-image --image-id ami-06863a26f4a103e6c**

#### Packer and VirtualBox

Of course, one of the many ways to use Packer is to build images for a local or on-premise virtualization solution including **VirtualBox**:

* Go to folder **M5/M5-1/4-packer-virtualbox**
* Examine the **centos-packer.json** file
* We can also examine the two files located in **scripts/** folder
* And the last one, that gives us the OS installation automatization portion of the process, which we can find in **http/ks.cfg**
* Start the build process with:

**packer build centos-packer.json**

* After a while the resulting file, in our case Vagrant box, will appear in **builds/** folder
* We can test it with the available **Vagrantfile** by executing:

**vagrant up**

* Then as usual we can create a session with

**vagrant ssh**

* And finally exit and destroy the VM:

**exit**

**vagrant destroy**

If we want to automate the publication of the box to the **Vagrant Cloud**, we must include this part as last **post-processor** portion:

**{**

**"type": "vagrant-cloud",**

**"box\_tag": "shekeriev/centos7-epel",**

**"version": "{{user `version`}}"**

**}**

Then we must add this **variable** block in the beginning to provide and automate the versioning of the box:

**"variables": {**

**"version": ""**

**}**

Finally, we must set the environment variable **VAGRANT\_CLOUD\_TOKEN**. For the purpose, we must create a token first. This can be done by visiting: <https://app.vagrantup.com/settings/security>

Once, we are ready, we can execute the actual creation and publishing process with:

**packer build -var 'version=1.0' centos-packer.json**

#### Install Vault

Let’s install **Vault** locally. For this purpose, we must execute the following steps:

* Download the latest version or we can copy the link from <https://www.vaultproject.io/downloads.html>
* Open a terminal window or session and execute:

**wget** [**https://releases.hashicorp.com/vault/1.0.1/vault\_1.0.1\_linux\_amd64.zip**](https://releases.hashicorp.com/vault/1.0.1/vault_1.0.1_linux_amd64.zip)

* Unzip the binary file

**unzip vault\_1.0.1\_linux\_amd64.zip**

* Move it to a folder part of your **PATH** variable, for example:

**sudo mv vault /usr/local/bin/**

* Check if everything is okay:

**vault**

If would be nice to have auto complete, at least in the beginning, so we can achieve it by:

* Executing:

**vault -autocomplete-install**

* And then reload the session:

**exec $SHELL**

#### Run Vault in DEV mode

We should be ready to start Vault in development mode and start experimenting with it:

* The actual command is:

**vault server -dev**

* Wait a while and keep watching the output. Write down the following values **Unseal Key** and **Root Token**
* Open another terminal window or session and initialize the following variables:

**export VAULT\_ADDR='http://127.0.0.1:8200'**

**export VAULT\_TOKEN='<Root Token>'**

* Now we can ask for Vault’ status with:

**vault status**

#### Working with secrets

Now that we have a running instance of **Vault**, we can start exploring its basic functionalities:

* Let’s create our first secret:

**vault kv put secret/demo secret1=Password1**

* We can put more than one at the same time. Let’s add one more:

**vault kv put secret/demo secret1=Password1 secret2=Password2**

* Now, that we have two values stored, we can retrieve them with:

**vault kv get secret/demo**

* We can retrieve the data in **JSON** format as well:

**vault kv get -format=json secret/demo**

* There is a way to parse the **JSON** output on the fly and extract particular value:

**vault kv get -format=json secret/demo | jq -r .data.data.secret2**

* The same result could be achieved with:

**vault kv get -field=secret2 secret/demo**

* Let’s delete the secret with all its values

**vault kv delete secret/demo**

#### Using other secret engines

We can use multiple secret engines and multiple instances of one engine at the same time:

* To check some of the possible engines, we can execute:

**vault plugin list**

* And to see what engines we are using currently, we can execute:

**vault secrets list**

* Now, let’s mount two more instances of the kv engine (the same as in the secrets/):

**vault secrets enable kv**

**vault secrets enable -path=demo kv**

* And then we can check again the engines in use and their mount points:

**vault secrets list**

* Now, that we have two more kv engine instances mounted, let’s add few secrets:

**vault write kv/test value=demo**

**vault write demo/data test1=value1 test2=value2**

**vault kv put demo/data2 key1=secret1 key2=secret2**

* It appears that we can store data with two similar commands. Now if we ask for the secrets:

**vault list kv**

**vault list demo**

* Of course, we can ask for the values as well:

**vault read demo/data2**

**vault kv get -format=json demo/data**

* Now, we can disable or unmount both secrets engines:

**vault secrets disable kv/**

**vault secrets disable demo/**

* And then we can check:

**vault secrets list**

#### Dynamic secrets (and AWS)

Dynamic secrets differ from what we have seen so far, the do not exist until they are read. We will experiment with the AWS engine:

* First, we must enable it:

**vault secrets enable aws**

* Then, before we continue, we must create a dedicated user (for example **vault**) in our AWS console with just **Programmatic Access** and **IAMFullAccess** policy attached. Of course, we must write down the **Access Key ID** and **Security Key ID**
* Then we will use the data to configure the aws engine:

**vault write aws/config/root access\_key=AKIAJTW4WX4UQ2CRSWCQ secret\_key=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**

Now that we have the initial configuration, we can create a role:

* Create a file **aws-ec2-role.json** with the following content:

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Action": ["ec2:\*"],**

**"Effect": "Allow",**

**"Resource": ["\*"]**

**}**

**]**

**}**

* Now create or store the role:

**vault write aws/roles/my-ec2-role credential\_type=iam\_user policy\_document=@aws-ec2-role.json**

* Now, that we have everything setup, we can generate a secret with:

**vault read aws/creds/my-ec2-role**

* We should write down the **lease\_id** as it will be needed, for example if we want to renew or revoke the secret
* If we return to the **Users** list in the **IAM** section of our **AWS** console, we should see a record matching the new secret we just created
* Current secret has a **lease\_duration** of **768 hours**, but we can configure this parameter with:

**vault write aws/config/lease lease=5m lease\_max=24h**

* Now we can create one more secret:

**vault read aws/creds/my-ec2-role**

* This one will be with the new value for **lease\_duration**
* Let’s revoke both secrets:

**vault lease revoke aws/creds/my-ec2-role/2pUGm3ixhwqeSI3C2Zu2Dc0V**

**vault lease revoke aws/creds/my-ec2-role/5h1S6igg1WbkzT6m7Q6CpSjf**

* If we return to the console, we will indeed see that both secrets are no longer there

#### Navigate the paths

Each secrets engine has its own structure, which we could not know upfront. It is good to know that there is a way to explore the structure with the internal command **path-help**

* Let’s check the situation with **secret/**

**vault path-help secret/**

* We can go even further

**vault path-help secret/config**

* Now compare the output of the first command to this:

**vault path-help aws/**

* Here we can explore for example this:

**vault path-help aws/config/lease**

* And why not read the current set of settings:

**vault read aws/config/lease/**

#### Authentication, Tokens, and Policies

When we start Vault in development mode authentication with tokes is enabled and it is the only way of authentication method. Of course, there are other options available, but currently we will stick with it. Let’s play a bit:

* Let’s create a new child token of our current token (root):

**vault token create**

* We can login with the new token

**vault login s.1x30j5G372iuE5zC0FPu9wgi**

* We will see a warning that there is an environment variable set
* Let’s login back with the initial token

**vault login s.88rZrDuEfWDGPjkz1PvLyp42**

* And revoke the one we created earlier:

**vault token revoke s.1x30j5G372iuE5zC0FPu9wgi**

Once a user is authenticated, usually he or she must be authorized for certain actions. This is handled by policies. There are two built-in policies – root and default:

* We can check what policies exist:

**vault policy list**

* Let’s examine the default policy:

**vault policy read default**

* Now, we can check the contents of the **vault-policy.hcl** file
* If we have some doubts about the structure and formatting of the file, we can execute:

**vault policy fmt vault-policy.hcl**

* We can upload the policy under the **my-vault-policy** name with:

**vault policy write my-vault-policy vault-policy.hcl**

* And we can ask again for the list of available policies and the contents of our new policy:

**vault policy list**

**vault policy read my-vault-policy**

Let’s check if the policy is working:

* First, we will create a token linked to the policy

**vault token create -policy=my-vault-policy**

* Now login with the new token, but before that let’s unset the environment variable:

**unset VAULT\_TOKEN**

**vault login s.5ULGc96cBqowZt9gPziZCg6o**

* Try to create one allowed secret and one not allowed:

**vault kv put secret/bar robot=beepboop**

**vault kv put secret/foo robot=beepboop**

* Login again with the initial root token

**vault login s.88rZrDuEfWDGPjkz1PvLyp42**

* Now revoke the other token

**vault token revoke s.5ULGc96cBqowZt9gPziZCg6o**

#### Vault Web UI

When started in development mode, Vault provides a web interface:

* Open browser tab and navigate to <http://127.0.0.1:8200/ui>
* Then use the default root token to log in
* Take a time to explore the available functions, try to do some of the previous tasks

## Part 2: Consul

#### Install Consul

There are multiple ways to install **Consul** under Linux, but for the purpose of our exercise we will use:

* Go to <https://www.consul.io/downloads.html>, check which is the latest version, and copy the link
* Open a terminal window / session and execute:

**wget** [**https://releases.hashicorp.com/consul/1.4.0/consul\_1.4.0\_linux\_amd64.zip**](https://releases.hashicorp.com/consul/1.4.0/consul_1.4.0_linux_amd64.zip)

* Then extract the zip archive

**unzip consul\_1.4.0\_linux\_amd64.zip**

* Move the file to a location, part of your **PATH** variable, or include the target folder in your **PATH**

**sudo mv consul /usr/local/bin/**

* Now check that everything is okay, by executing:

**consul**

#### Run Consul in DEV mode

We are ready to test Consul in a development mode. For this purpose, do the following:

* Open a terminal window and type

**consul agent -dev**

* Open a second terminal window and ask for the members of our Consul cluster:

**consul members**

* We can ask for more detailed information with:

**consul members -detailed**

* As other products in the portfolio, Consul offers **HTTP API**:

**curl localhost:8500/v1/catalog/nodes**

**curl localhost:8500/v1/agent/members**

* There is one more option for information – the **DNS interface**:

**dig @127.0.0.1 -p 8600 lenovo-e450-dz.node.consul**

Once we are done with the steps, we can return in the first terminal and stop the service by pressing **Ctrl+C**

#### Register a service

Services definitions can be stored as **JSON** files. Now let’s experiment:

* Go to folder **M5/M5-2/1/**
* Start **Consul**, but with a configuration folder as parameter:

**consul agent -dev -config-dir=./consul.d**

* Examine the only file in the **consul.d** folder – **web.json**
* Now, let’s use the DNS interface and ask questions about the service. First, we will ask about the A record:

**dig @127.0.0.1 -p 8600 web.service.consul**

* Then, we can ask for the port and the server address of the service:

**dig @127.0.0.1 -p 8600 web.service.consul SRV**

* Furthermore, we can filter services by tags:

**dig @127.0.0.1 -p 8600 php.web.service.consul SRV**

**dig @127.0.0.1 -p 8600 dev.web.service.consul SRV**

* Last, but not least, we can use the **HTTP API** to query **Consul** for information about services:

**curl http://localhost:8500/v1/catalog/service/web**

* Or for their health:

**curl http://localhost:8500/v1/health/service/web**

#### Connect, Services, and Proxies

* Go to folder **M5/M5-2/2/**
* Start **Consul**

**consul agent -dev -config-dir=./consul.d**

* Open another terminal and start the following simple echo service

**socat -v tcp-l:8181,fork exec:"/bin/cat"**

* Open third terminal window and test that the echo service is working:

**nc 127.0.0.1 8181**

* Then enter a string, for example **hello**, and press **Enter**
* Now press **Ctrl+C** to exit the **nc** utility
* Register the **socat** service with **Consul** by copying the prepared file to the configuration folder:

**cp socat.json consul.d/**

* Now force the **Consul** to reload its configuration

**consul reload**

* Let’s start the proxy process in the third terminal

**consul connect proxy -sidecar-for socat**

* Open a fourth terminal and create inline service dependent on the **socat**:

**consul connect proxy -service web -upstream socat:9191**

* Start a fifth terminal window and use the **nc** utility to check that the dependent service is working:

**nc 127.0.0.1:9191**

* Type a word and press **Enter**. Then hit **Ctrl+C**
* Return to the fourth terminal and hit **Ctrl+C** as well
* Being in the fourth terminal copy the **web.json** file to the configuration folder. This way we are going to register the service with **Consul** instead of using the inline approach, by using a configuration file

**cp web.json consul.d/**

* Reload the **Consul** service

**consul reload**

* Now, it is time to start the web proxy:

**consul connect proxy -sidecar-for web**

* Return to the fifth terminal window and check with **nc** that the setup is working:

**nc 127.0.0.1 9191**

* Type a word and press **Enter**. Then press **Ctrl+C** to close the **nc** utility
* Let’s create an intention that will deny connection between **web** and **socat** services

**consul intention create -deny web socat**

* And try again with the **nc** utility

**nc 127.0.0.1 9191**

* If we try to type something and hit **Enter**, we are presented with the command prompt again, which means that the intention is working and the connection between **web** and **socat** is denied as expected
* If we delete the intention:

**consul intention delete web socat**

* And try again:

**nc 127.0.0.1 9191**

* It must work
* Let’s clean up a little bit. Close all five terminal windows

#### Consul cluster

Typically, we will interact with Consul clusters, and not with single node installations. So, let’s create one:

* Go to folder **M5/M5-2/3**
* Examine the **Vagrantfile**
* Create the infrastructure with:

**vagrant up**

* Once the machines are up, we can establish a ssh session to the first one:

**vagrant ssh n1**

* And start the **Consul** service:

**consul agent -server -bootstrap-expect=1 \**

**-data-dir=/tmp/consul -node=agent-one -bind=172.20.20.10 \**

**-enable-script-checks=true -config-dir=/etc/consul.d**

* Open second terminal, and connect to the second node:

**vagrant ssh n2**

* Then start again the **Consul** service, but this time with:

**consul agent -data-dir=/tmp/consul -node=agent-two \**

**-bind=172.20.20.11 -enable-script-checks=true -config-dir=/etc/consul.d**

* Open third terminal and check how many nodes sees the Consul process, running on the first node:

**vagrant ssh -c 'consul members' n1**

* Do the same for node two:

**vagrant ssh -c 'consul members' n2**

* Okay, let’s instruct the first node to connect to the second node:

**vagrant ssh -c 'consul join 172.20.20.11' n1**

* And check again how many nodes sees the first node:

**vagrant ssh -c 'consul members' n1**

* We can use either the DNS interface or HTTP API to ask for additional information:

**vagrant ssh -c 'dig @127.0.0.1 -p 8600 agent-two.node.consul' n1**

**vagrant ssh -c 'curl localhost:8500/v1/agent/members' n1**

Let’s leave the cluster running. Later we can return here and clean up everything:

* We can shut down the services – in each session hit **Ctrl+C**
* Then close the connections by executing

**exit**

* And finally, clean up by executing

**vagrant destroy --force**

#### Health checks

Let’s continue by adding the checks:

* Go to folder **M5/M5-2/4/**
* Examine the two files
* Now return to the **M5/M5-2/3/** folder
* Establish ssh connection to node 2

**vagrant ssh n2**

* Execute this command to create a check:

**echo '{"check": {"name": "ping",**

**"args": ["ping", "-c1", "google.com"], "interval": "30s"}}' \**

**>/etc/consul.d/ping.json**

* And this, to create a service:

**echo '{"service": {"name": "web", "tags": ["rails"], "port": 80,**

**"check": {"args": ["curl", "localhost"], "interval": "10s"}}}' \**

**>/etc/consul.d/web.json**

* Then trigger configuration reload

**consul reload**

* Now, use the **HTTP API** to get list of all services marked as critical

**curl http://localhost:8500/v1/health/state/critical**

* Then we can use the DNS interface, but because the service is failing, no data will be returned

**dig @127.0.0.1 -p 8600 web.service.consul**

#### Consul KV data

As an addition to the service discovery Consul provides a KV data store. Let’s explore it, it should be looking familiar:

* Being on the second node, insert few values

**consul kv put redis/config/minconns 1**

**consul kv put redis/config/maxconns 25**

**consul kv put -flags=42 redis/config/users/admin abcd1234**

* Now, switch to the first node, and get the value of one key

**consul kv get redis/config/minconns**

* Get detailed information about one key

**consul kv get -detailed redis/config/minconns**

* Get all keys and their values

**consul kv get -recurse**

* We can delete one key

**consul kv delete redis/config/minconns**

* Return to node two, and all keys

**consul kv delete -recurse redis**

#### Consul Web UI

Now, let’s try the nice Web UI. For this purpose, we must stop each service, and start them with an extended set of parameters:

* Return to the initial connection to n1 and press Ctrl+C. Then execute this command:

**consul agent -server -bootstrap-expect=1 \**

**-data-dir=/tmp/consul -node=agent-one -bind=172.20.20.10 \**

**-enable-script-checks=true -config-dir=/etc/consul.d -ui -client=0.0.0.0**

* Go to the initial connection to n2 and press Ctrl+C. Start it again, but with:

**consul agent -data-dir=/tmp/consul -node=agent-two \**

**-bind=172.20.20.11 -enable-script-checks=true -config-dir=/etc/consul.d -ui -client=0.0.0.0**

* Then re-join the second node to the first one with:

**vagrant ssh -c 'consul join 172.20.20.11' n1**

* Ask for the status:

**vagrant ssh -c 'consul members' n1**

* Open a browser tab on your host and navigate to: <http://localhost:8501>
* Explore the UI
* Repeat some of the previous tasks and watch them in the UI

## Part 3: Nomad

#### Install and running Nomad

We will use a **Vagrantfile**, provided by **HashiCorp**. Let’s get started:

* Go to folder **M5/M5-3/1/**
* Examine the **Vagrantfile** and start the machine with

**vagrant up**

* Then establish a ssh session to the machine

**vagrant ssh**

* Now check all the installed components:

**nomad version**

**consul version**

**docker info**

* Now, that everything is working, we are ready to start nomad service in development mode:

**nomad agent -dev**

* We must open a second terminal and establish another session to the machine:

**vagrant ssh**

* Let’s see the registered nodes (we expect only one of course):

**nomad node status**

* We can display the list of all members of the cluster with:

**nomad server members**

* And ask for even more information:

**nomad server members -detailed**

* Then, as with other HachiCorp’ services, we can press Ctrl+C to stop the nomad service. We can skip it for now

#### Run a Job

While still in the Vagrant machine, we can create our first job:

* If we have a job file, we can provide it to the Nomad cluster, but we can also ask Nomad to produce one sample job for us:

**nomad job init**

* Let’s examine the job’s definition
* And then register it with Nomad

**nomad job run example.nomad**

* Then, we can ask for its status

**nomad status example**

* And check in Docker as well:

**docker container ps**

* We can request more information about the allocation:

**nomad alloc status 3bb6ce6d**

**nomad alloc status -verbose 3bb6ce6d-ce24-939d-999a-7e3b4a1bd684**

* There is a way to ask for logs of the task:

**nomad alloc logs 3bb6ce6d-ce24-939d-999a-7e3b4a1bd684**

* What if we need more copies of the task group? Let’s edit the job definition, and set it to 3
* Before we apply the changes, we can ask for the plan of their application (just like with Terraform):

**nomad job plan example.nomad**

* To apply the changes, use the command hint shown by the planning stage:

**nomad job run -check-index 18 example.nomad**

* If we ask for the status, we will see that there are three allocations:

**nomad status example**

* Now, let’s change the image to **redis:4.0** and save the job. Then ask for the plan again:

**nomad job plan example.nomad**

* And apply it of course:

**nomad job run example.nomad**

* Let’s check for the status:

**nomad status example**

* We can repeat asking for the status to monitor the upgrade process
* Once all task groups are updated, we can stop the job and ask for the status again

**nomad job stop example**

**nomad status example**

* We can check how are the things from **Docker** point of view:

**docker container ps -a**

**docker image ls**

#### Nomad cluster on a single machine

Let’s expand our configuration with multiple agents each with its own role:

* Go to folder **M5/M5-3/2/**
* Start the machine (the **Vagrantfile** is the same)

vagrant up

* Meanwhile, examine the three **HCL** files
* Once the machine is up, copy **\*.hcl** and then establish an **SSH** session to it

**scp -P 2222 \*.hcl vagrant@localhost:.**

**vagrant ssh**

* Start the server process:

**nomad agent -config server.hcl**

* Open second session to the machine and start the first client

**vagrant ssh**

**nomad agent -config client1.hcl**

* Open third session and start the second client:

**vagrant ssh**

**nomad agent -config client2.hcl**

* Now open a fourth connection and check the status

**vagrant ssh**

**nomad node status**

* Let’s create the sample job definition again, but this time a condensed one:

**nomad job init -short**

* Open the file and in the **group** section add **count = 5**
* Now run the job

**nomad job run example.nomad**

* And let’s check for the status:

**nomad status**

**nomad status example**

#### Nomad Web UI

As with the other products, covered so far, here there is a Web UI also:

* Open a browser tab on your host machine, and navigate to <http://localhost:4646/ui>
* Explore the interface
* Stop the job created in the previous section

#### Nomad cluster with multiple machines

We can try a scenario in which we have multiple machines – three servers and one client:

* Go to folder **M5/M5-3/3/** and examine the **Vagrantfile**
* Initiate the infrastructure creation
* Then explore both folders – consul and nomad
* Examine the **script-steps.txt** file as well
* Once the environment is ready start nine sessions – three for the **nomad-server1** and two for others
* Follow the steps describe in the text file
* Open the Consul Web UI by visiting <http://localhost:21001>
* Open the Nomad Web UI by visiting <http://192.168.50.11:4646>