**REPORT**

Task 1

Automated System Development Using Vagrant-Ansible

Prepared: Seyfullah Kaya

# Purpose and Requirements

Our purpose in designing the system is to create a lab environment that will be used in future tasks and that is automatically started up. The reason why we did not try the tasks on baremetal is that our system may be damaged while performing some dangerous operations. Therefore, it would be best to run it in a virtual environment in terms of both our time and system security. However, running the lab environment may still cause us trouble. Therefore, I designed the entire lab environment that we created using ansible and scripts to be started up with just one command.

We need some requirements to get the system up and running. I created the test environment on bare metal debian. Vagrant, libvirt and vagrant-libvirt plugins must be installed on bare metal. The “start\_all.sh” script in the main directory I created checks the requirements in order and automatically installs them if there is anything missing. You can also perform the necessary installations manually. The stages will be available on github. I entered the config settings for two virtual machines in a single Vagrantfile. Since our Windows machine will be nested-virt, I allocated 16GB RAM and 4 CPU cores. Since my Ubuntu machine will work entirely for ansible management, I kept its features minimal. I assigned static IP addresses to both virtual machines (libvirt\_winvm and libvirt\_ubuntu). I will present all the configs to you in the following lines of the report. You can change the features from the relevant places according to your own needs.

Another requirement for the system is that I could not give a static IP address to my dualboot debian virtual machine that I automatically set up with the preseed file I wrote. It preferred to assign this IP address with DHCP every time, and because of this problem, I could not manage my dual-boot debian virtual machine that I set up with my ansible machine and I could not load the requirements to that virtual machine and start it. Therefore, I wrote the management settings directly for the host machine with ansible. In our “/libvirt/scriptsdeb/hosts.ini” file, you need to fill in the authorization information from the part written as “[baremetal:vars]”. Thanks to this information, our ansible machine will be able to communicate with the host machine without any problems and complete the necessary automation without any problems. I will explain the system’s yml playbooks and all the work they do in the next section completely and clearly.

# Operating the System and Its Functioning

To run the system, we first need to download the files from github. Then our *“libvirt/start\_all.sh”* script will start the entire system.

* Let's access the *"/libvirt"* folder with the terminal and type *"./start\_all.sh"* into the terminal. This command will run our start\_all.sh script and perform all the steps automatically. So what does this script do?
  + In the first stage, our script checks the requirements in the system (vagrant, libvirt, vagrant-libvirt plugin). If these software are not available, it automatically performs an installation.
  + It sets the internet interface we use in the system. I have communicated all the machines in the system with each other over the 192.168.121.0/24 network. If this interface does not exist, it creates it.
  + After the requirements checks, he enters the necessary commands to run our virtual machines. First, we need to run our Windows virtual machine with the *"vagrant up winvm"* command and set its content. Once this process is complete, he runs our Ubuntu machine with the *"vagrant up ubuntu –provider=libvirt"* command.
  + While running our Windows machine, we perform the following operations on our Windows machine through the scripts I wrote in the Vagrantfile;
  + First, it statically gives our machine the IP address 192.168.121.130 and pulls the ConfigurationRemotingForAnsible.ps1 script found in github to our Windows machine so that it can be managed with Ansible and runs this script.
  + Then, it runs our *“/libvirt/scriptswin/install\_qemu.ps1”* script by sharing it with our virtual machine via rsync. This script installs qemu on our virtual machine and activates the necessary windows features (Microsoft Hyper-V, Hypervisor Platform, Virtual Machine Platform, WSL). Thanks to the Hyper-V feature, it may have activated whpx, which is the hardware accelerator in QEMU’s windows.
  + Finally, it runs the *“/images/download.ps1”* script that we shared with rsync to my virtual machine to download our previously prepared OMV.vmdk disk. This script first installs python on our virtual machine and saves it to the system path. Then, using pip that he downloaded with python, he installs gdown and using this gdown, he downloads the disk.vmdk disk containing the OMV boot from the link, so that it can be changed in the next pages of the reports. After this download, a second 10G disk is created as disk1.vmdk in the folder of the folder using qemu-img. We will use these two disks to run our OMV virtual machine..
* After the above operations, our Windows virtual machine is successfully activated to be managed with Ansible.
* Then our *“start\_all.sh”* script performs its last function and wakes up the Ubuntu virtual machine with the *“vagrant up ubuntu –provider=libvirt”* command. When this virtual machine wakes up, it gets the 192.168.121.20 IP address statically and automatically runs the *“/libvirt/scriptsdeb/req.sh”* file. This file is;
  + Ubuntu CLI updates the packages of our virtual machine and installs ansible and winrm to perform management.
  + Then, using the information in *the “/libvirt/vagrant/scriptsdeb/hosts.ini”* file, it applies the following playbooks in order;
    - The *“/libvirt/vagrant/scriptsdeb/shrinkdisk”* playbook shrinks a 20GB area from the 60GB main disk in our Windows virtual machine. In addition, this area is formatted by separating it from exFAT using the OMV name and D label. Formatting the disk as exFAT can be selected because Windows, Linux and macOS can identify and strengthen the exFAT format*.*
    - The *“/libvirt/vagrant/scriptsdeb/copy\_disks.yml”* playbook copies the disk.vmdk and disk1.vmdk disks we downloaded to the D drive we created in the previous step.
    - The *“/libvirt/vagrant/scriptsdeb/start\_vm.yml”* playbook creates two files. One of them is created in the *“c/vagrant\_vm\_boot”* folder, while the other one is created in the startup folder. Our first file contains the qemu boot config settings. It selects whpx as the accelerator, assigns 4GB of RAM, and gives two disk entries. One of them is disk.vmdk copied to disk D and this disk is selected as our boot disk. The second one is disk1.vmdk disk that we created and copied to disk D and this disk works as an extra storage disk. We do port forwarding in order to access the virtual machine. In other words, we can connect to the OMV web interface by typing the link above into the browser from any device on the same network, *“<virtual\_machine\_ip>:8080”.* Finally, this playbook creates a bat file so that our virtual machine can be opened every time Windows is started. This bat file is copied to the startup folder and this bat file runs our start\_vm.ps1 file with administrator permissions. In this way, every time our Windows machine starts, it starts our OMV virtual machine with the qemu startup settings we have specified and allows us to access it without any problems.
    - The playbook *“/libvirt/vagrant/scriptsdeb/close\_windows.yml”* closes our Windows virtual machine to free up disks as our automation processes with Windows are finished. After this process, our request file enters a 30 second waiting period.
    - Thanks to the *“/libvirt/vagrant/scriptsdeb/run\_preseed.yml”* playbook, we run the *“/libvirt/preseed/auto\_debian\_install.sh”* script on the host machine. This script is;
      * It offers the location of the Preseed file by opening http server using python. It opens a new virtual machine with libvirt. This virtual machine it opens takes two disks so that we can make the necessary settings. It takes the debian.qcow2 disk that we created in Vagrantfile and run with Windows as the vda disk. Instead of creating this disk normally, I tried to create it by shrinking it from the raw disk in vagrantbox and install Debian there automatically. However, each time, it installed Debian on the entire disk and my Windows disk exploded. As a solution to this, I preferred to add a second disk and install Debian on it. In addition to the vda disk, it takes my libvirt\_winvm.img disk, which also contains my OMV disk, as the second disk, namely vdb. It sets the name of the new virtual machine it opens as *“debian-in-windows”* and deletes it if such a virtual machine has been created before. Then, we enter the required parameters in our virtual machine settings and start the automatic installation by accessing our preseed file via the http server.
    - After our automatic debian installation is completed, it runs the playbook *“/libvirt/vagrant/scriptsdeb/run.yml”*. This playbook runs the expect script *“/libvirt/static\_ip/set\_ip\_console.sh”* by accessing it from the host machine. This expect script is;
      * It provides access to the Debian virtual machine we created with the command *“virsh console debian-in-windows”* from the host machine. I resorted to this method because I could not give a static IP to the machine we created after a long effort via preseed. Thanks to this, I can connect to the virtual machine I created directly from the host machine with the console and access it with ansible by adding the IP address 192.168.121.145 to *“/etc/network/interfaces”.*
    - Thanks to the *“/libvirt/vagrant/scriptsdeb/install\_qemu.yml”* playbook, qemu is installed in the Debian virtual machine whose IP address we changed directly from the host machine with expect.
    - Thanks to the

*“/libvirt/vagrant/scriptsdeb/start\_vm\_when\_reboot\_debian.yml”* playbook, we first save the OMV disk (originally vdb2) in the second disk that we gave as a parameter to the *“/etc/fstab”* file in order to mount it permanently. Then, with this information and the necessary parameters, we enter the necessary config settings to start our QEMU virtual machine as a service. Thanks to these settings, our disk.vmdk file (the OMV disks in the disk we mounted) is added as a boot disk, while my disk1.vmdk disk is added to the system as an additional disk. It uses kvm as a hardware accelerator and activates port forwarding. It uses ports 8080 for http, 8443 for https, and 2222 for ssh. By using the *“<virtual\_machine\_ip>:8080”* link, we can access the OMV web interface from any device connected to the same network without any problems. Then, we record the commands required to activate this service at every Debian startup under the name qemu-vm.service.

* + - The boot settings of our virtual debian machine were completed without any problems thanks to the playbooks above. Thanks to the *“/libvirt/vagrant/scriptsdeb/change\_vda.yml”* playbook, we update the vdb disk name in the *“/etc/fstab”* file that we set up so that we do not have access problems from the second virtual machine and can make the settings, to vda. In this way, since the disk will appear as vda in our main virtual machine, we do not have any problems with booting.
    - The *“/libvirt/vagrant/scriptsdeb/delete\_enp1s0.yml”* playbook accesses the Debian virtual machine we created, deletes the static IP configurations in the *“/etc/network/interfaces”* file, and assigns the IP address 192.168.121.145 for the enp8s0 interface on our main virtual machine. Then, it shuts down the virtual machine without waiting for a response.
    - *“/libvirt/vagrant/scriptsdeb/enable\_dualboot\_on\_baremetal.yml”* accesses the playbook host machine and accesses the configuration settings of our main virtual machine, first marking the second disk as bootable and then enabling the boot menu. After these operations, it undefines and redefines the virtual machine. In this way, our main virtual machine gives us a choice every time it is restarted. If we select the first disk, Debian starts, if we select the second disk, Windows starts. Whichever option we choose, they start a qemu virtual machine with the OMV disk they have access to together. We can access this OMV virtual machine from any device on the same network using port forwarding*.*

As a result of the tests, the time required for the lab environment to become operational is approximately 1 hour. The table below contains the IP information, user information, and access information of the virtual machines.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Username | Password | Virt Level | IP Address |
| Windows | vagrant | vagrant | 1 | 192.168.121.130 |
| Debian dualboot | User:user  Root:root | user:userpass  root:rootpass | 1 | 192.168.121.145 |
| Ubuntu Ansible | vagrant | vagrant | 1 | 192.168.121.20 |
| Openmediavault | CLI:root  Web:admin | CLI:1647  Web:sanbox | 2 | Virtual OS IP |
| Baremetal | root | 1647 | - | 192.168.121.1 |

The Virt Level information given in the table above is the virtualization layer information. Baremetal is our main machine. Windows, Debian Dualboot, Ubuntu Ansible is the first level virtualization. Openmediavault is the second level virtualization. (Nested-virt, virtual machine within virtual). Openmediavault uses the IP address of the virtual machine it is in.

* [Click](https://drive.google.com/file/d/1Xf_O8pprBlkvgMcjBodDnoYdFOh6JFC9/view?usp=sharing) for OMV disk download link
* [Click](https://github.com/ReqwerT/labfortasks) for Github repo link

# Results

As a result of the operations, two virtual machines were started up completely automatically using vagrant. After our Windows machine was ready, our Ansible virtual machine started up and managed our Windows machine and fulfilled the requirements completely automatically. Then, our Ansible virtual machine interacted with our main machine and installed the Ubuntu operating system on the Debian.qcow2 disk completely automatically using the virtual machine it opened with libvirt on the main machine and the preseed file it served from the main machine. Since the Ubuntu virtual machine assigns its IP address via DHCP, it communicated with the main machine and automatically assigned a static IP address to the Ubuntu machine. As a result of the IP address assignment, Qemu was automatically installed on the Ubuntu machine that became manageable with Ansible and the disk was mounted. The operations required for our Qemu virtual machine to start as a service with the correct parameters at every reboot were applied to the Debian virtual machine with Ansible. After the IP configurations and connection options were automatically updated, the second machine created was automatically shut down and access was provided to the main machine, the boot menu was activated on the first virtual machine and the Debian disk was marked as bootable. As a result of the operations, our lab environment was automatically brought to life. The tests performed show us that this environment was ready in approximately 1 hour. Regardless of the selected operating system, access to the common exFAT OMV disks was performed without any problems in every operating system that was opened, and each time the virtual machine was restarted, the OMV virtual machine opened within the virtual without any problems, and the access tests were successful in both operating systems.

## Computer specifications on which the test was performed

|  |  |
| --- | --- |
| Model | Asus  fx706li-hx199 |
| CPU | Intel Core i5 10300h |
| RAM | 32Gb DDR4 |
| DISK | 480gb M2 SSD |
| OS | Bare Metal Debian |
| Vt-x | Yes |