# Ansible Handson

### Convenient Vagrant commands:

### *vagrant up*: brings up all machines defined in the Vagrantfile of the current directory *vagrant ssh <name>*: connects to the VM *vagrant reload <name>*: forces a reload of the VM, useful to force new configuration to be loaded *vagrant halt <name>*: stops the VM

### *vagrant provision <name>: provisions the VM using the provision configuration in the vagrantfile*

A warning beforehand. Read the instructions **carefully**, every step is important.

## Setting up:

First off, make sure your Ansible host VM is installed and configured. The VM supplied by Vagrant does not have Ansible installed so you must install it yourself. The provided Vagrantfile will provide you with a basic VM, run it using *vagrant up*. Once it is up and running connect to it (vagrant ssh ansiblehost). The set of commands below will install Ansible and check the version to see if the installation is successful.  
 *sudo yum -y install ansible  
ansible --version*

For this tutorial we will disable Ansible host key checking, it will make everything unnecessarily difficult. Edit the Ansible config by executing *sudo vim /etc/ansible/ansible.cfg*. Uncomment #host\_key\_checking = False and save the file.

N.b., remember that you can write your file inside vi using :w and quit using :q.

Congratulations, Ansible is now configured and ready to use.

**Time to create another VM** that we will use to host our Apache webserver. Let’s call it apachewebserver. Paste below code into the Vagrantfile, at the bottom, before the last **end**. Mind the whitespace!

*config.vm.define "apachewebserver" do |apachewebserver|  
 apachewebserver.vm.hostname = "apachewebserver"  
 end*

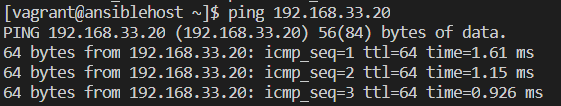
Again, type *vagrant up* in your command line (make sure you execute this command on your own command line, **not** inside the “ansiblehost” virtual machine). Vagrant will now create a new VM to host your Apache webserver. Notice how Vagrant leaves your ansiblehost alone because it’s already up and running.

Now we have two VMs but no network connecting them together. This way it will be impossible to have our Ansible machine do anything to our Apache server. Remember that the ansiblehost needs to be able to connect to any VM it wants to configure through SSH.  
Time to set up a network! Paste the following lines into the Vagrantfile, into their respective VM blocks. Make sure to match the indentation! It matters!

*ansiblehost.vm.network "private\_network", ip: "192.168.33.10"*

*apachewebserver.vm.network "private\_network", ip: "192.168.33.20"*

Now, type *vagrant reload* to make sure Vagrant sets up your network. Once both VMs have restarted ssh to one of the VMs and check the ip address with ip addr. The private network should show up! Ping the other server (using the IP address assigned in the Vagrantfile) to check if the connection works fine. It should work like this:



Now it is time to configure the Ansible hosts file inside of the ansiblehosts VM. Log in to the ansiblehosts VM if you are not logged in (vagrant ssh ansiblehost). Create a new hosts file /etc/ansible/hosts

*sudo mkdir /etc/ansible && sudo touch /etc/ansible/hosts && sudo vi /etc/ansible/hosts*

And fill it with the following:

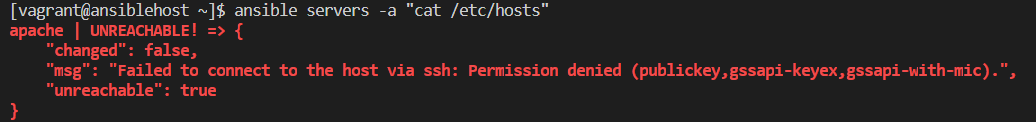
*[servers]  
apache ansible\_host=192.168.33.20*

N.b., remember that you can write your file inside vi using :w and quit using :q.

Let us inspect what this means. There is a group of hosts called servers, inside this group there is a server 192.168.33.20 which has the alias apache. OK! Ansible on the ansiblehost will now know that if you want to run Ansible commands on the group servers it needs to connect to 192.168.33.20 (just one server at this moment).

Time to run our very first Ansible commands to check if it all works fine. Issue the following command on ansiblehost:

*ansible servers -a "cat /etc/hosts"*

Ouch! That did not go as planned. N.b., it is possible that you will be asked for confirmation before connecting, since the apache server is not in the known hosts. You can safely add it to the known hosts.

It appears that the apachewebserver does not allow us to connect using SSH. Since Ansible uses SSH that means we need to set up a key, so that it will accept the connection. On the ansiblehost server run the following command (leave the password empty and save it in the default directory: /home/vagrant/.ssh/id\_rsa)

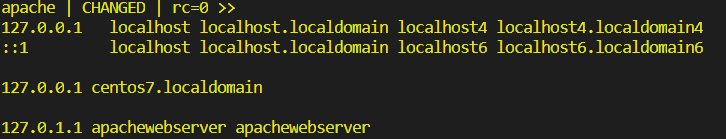
*ssh-keygen -t RSA*

Now copy the contents of /home/vagrant/.ssh/id\_rsa.pub to the clipboard. We will copy the key to our apachewebserver. **Make sure you copy the contents of the public key, not the private key.** Log out of the ansiblehost and log in to the apachewebserver. Paste the contents of your key inside ~/.ssh/authorized\_keys. Alright, log out and back in to our ansiblehost server; try to ssh 192.168.33.20. It should work fine. If it doesn’t check if you pasted the public key into authorized\_keys of the apachewebserver, if you pasted the private key it will not work.

Now that we have verified that the SSH connection works, it is time to run your very first Ansible command. This will be an Ansible ad hoc command.

*ansible servers -a "cat /etc/hosts"*

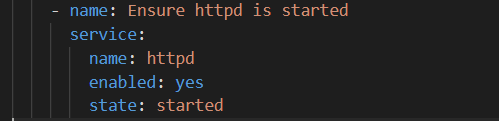
If all went well you should see

Congratulations! You managed to run your very first Ansible command, which ran “cat /etc/hosts” on the apachewebserver VM.

## Your first Playbook: provisioning the apachewebserver

It is time to write your first Playbook. Ansible ad hoc commands (like the one we just executed: ansible servers -a "cat /etc/hosts") are very useful when you need to quickly run some command(s) but when you want to automatically provision or deploy you will need to use a Playbook. A Playbook is basically a list of ad hoc commands, in human readable format. A Playbook is run against a predefined set of hosts and as one specific user, usually a “deployment” user. In this tutorial we will have Ansible become root during deployments. This tutorial is quite long.  
Remember that you can check the validity of your Playbook at any time not only through the Ansible Visual Studio Code extension, but also by running *vagrant provision ansiblehost*. Doing so regularly will alert you to mistakes early on, instead of having to fix error after error after completing the Playbook.

The Playbook must contain a certain set of tasks to be executed. An Ansible task consists of a command that is to be run using a certain Ansible module. For example the following task will ensure that the httpd service is enabled and started. Notice the syntax. First is the (optional) name of the task. Second is the name of the Ansible module, in this case the service module. Third is a dictionary (an array) specifying what the module should ensure. In this case it will ensure that the httpd service (our Apache webservice) is enabled and started. See the Ansible documentation on modules for more information: <https://docs.ansible.com/ansible/latest/user_guide/modules.html>.



Ansible has a lot of modules, after a while it becomes second nature to simply Google for an Ansible module because it’s difficult to know them all by heart. Why use Ansible modules though? Ansible can run commands simply by executing shell commands like so:

Ansible modules ensure idempotency. A shell command would simply execute that command, a module will ensure that the server reaches a certain state. Whenever possible you should be using modules in your Playbook, not bare commands.

Now, in your main directory (the directory that contains your Vagrantfile, on your own PC, not inside any VM) create a file called playbook.yml. This will be your Playbook. When in doubt, check the lecture slides for the exact syntax or ask the instructor! The lecture slides contain excellent examples on how to do everything from here on forth. I very highly recommend to complete the rest of the tutorial with the lecture slides as a guide.

**For your first exercise, edit the playbook.yml so that:**

* the playbook should run against all hosts and become root.
* the playbook uses the Ansible yum module to install Apache
* the playbook uses the Ansible service module to ensure that Apache is started after installation.

If you get stuck, remember that Google is a useful tool. The Ansible documentation is very good and should help you out. If you are still stuck, ask the instructor for help!

Once you have your Playbook it is time to run it and see if it will work. Remember, the goal is to have the ansiblehost machine execute the Playbook against the Apache webserver. At this moment your Playbook is not inside of the ansiblehost VM, and we would like to keep it that way (so that we can edit the Playbook inside of Visual Studio Code!).

We can make Vagrant run Ansible commands inside of the ansiblehost. To do so we must edit the Vagrantfile. Again, it is also possible to ssh to the ansiblehost machine and make that run the Playbook but that would require us to place the Playbook inside of the VM. Using the Vagrantfile is much more simple. Add the following to the ansiblehost configuration. Again mind the indentation!

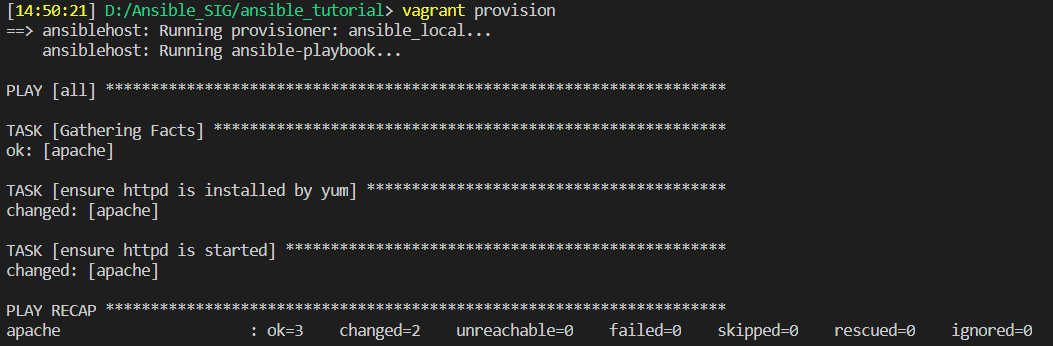
*ansiblehost.vm.provision :ansible\_local do |ansible|  
 ansible.playbook = "playbook.yml"  
 ansible.verbose = false  
 ansible.install = false  
 ansible.limit = "all" # or only "servers" group, etc.  
 ansible.inventory\_path = "inventory"  
 end*

Notice that this needs an inventory file (*ansible.inventory\_path = "inventory"*). For now we can simply copy the contents of our /etc/ansible/hosts file (inside the ansiblehosts VM) into a file called inventory, in the main directory of Vagrant (on your own PC). Remove the file /etc/ansible/hosts from the ansiblehosts VM, it will no longer be necessary.

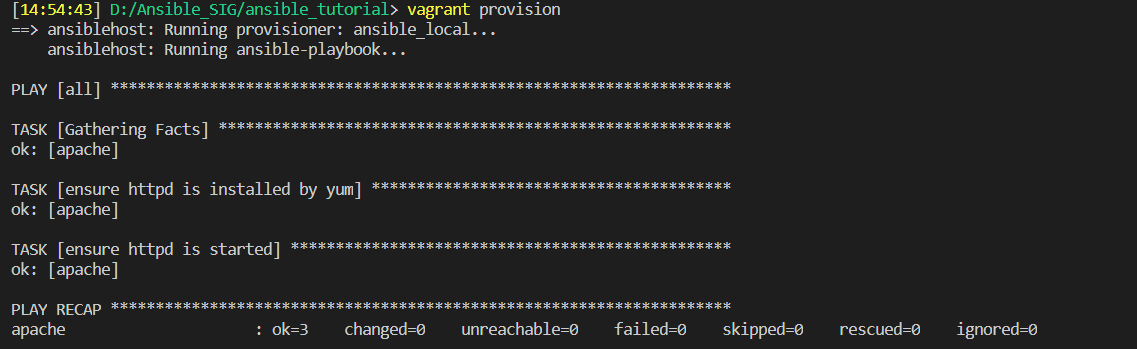
Now in your terminal, run the command

*vagrant provision*

This will use Vagrant to provision the ansiblehost. Now because our definition of provisioning the ansiblehost has it run a Playbook, it will run the Ansible Playbook against the apachewebserver. When successful, it should show something like below:



Inspect the output. First Ansible gathers facts (it connects to the host and gathers the hostname). Next it shows the tasks that are run; including the custom name we have defined in the Playbook. It shows whether the tasks were successful or not. Notice that Ansible also reports that the task has changed the server. If you were to rerun the Playbook, with Apache already installed, up and running, you will find that Ansible simply checks the server and reports that the server is in “OK” state.



This is the idempotence that makes Ansible excellent. It will not waste time running commands that do not need to be run. Ansible gets a server in a wanted state, and if it already is in that wanted state, it will not change anything.

We should test whether Apache actually works. It should show a basic page. At the moment port 80 of our Apache webserver cannot be reached; we need to forward a port to open it up. Add the following line to the apachewebserver configuration in the Vagrantfile:

*apachewebserver.vm.network :forwarded\_port, guest: 80, host: 8080*

This will forward port 8080 on the host machine (your PC) to port 80 of the VM. *vagrant reload apachewebserver* to enable the new configuration and visit 127.0.0.1:8080 to see your very own Apache webserver, provisioned by Ansible.

## Getting more advanced: adding a simple React application

Now that we have an Apache webserver it’s time to put it to good use. We will create and provision a new VM that will host a simple React application. No more hand-holding, this time you will have to figure out the code for yourself.  
Create a VM that is named reactserver, connected to the private network with IP 192.168.33.21. It should not be reachable from the outside so no ports need to be forwarded. Don’t forget to copy your public key to your new reactserver’s ~/.ssh/authorized\_keys! Otherwise Ansible will not be allowed to connect.

Once you have the VM it’s time to adapt your Playbook so that will provision the reactserver. This is not as straightforward as it sounds! If you simply add the commands below the Apache webserver part it will install and start Apache on your reactserver as well as on your apachewebserver. This is not the goal! We want to install Apache only on apachewebserver, not on the reactserver. Hint: use Ansible roles!

To run our React application we need to install NodeJS on our reactserver. There are multiple ways to do this. You could certainly write your own role to install NodeJS however… there is a very nice repository with tons of Ansible roles that you can download and use! That repository is called **Ansible Galaxy**: [https://galaxy.ansible.com](https://galaxy.ansible.com/). Usually, if you want to have some piece of popular software installed (such as Java, Docker or NodeJS) a role exists for this in Ansible Galaxy. Using Ansible Galaxy will save you a LOT of time. I very highly recommend searching Ansible Galaxy before creating your own roles.

Search for a NodeJS role on Ansible Galaxy. I would recommend looking for roles created by “geerlingguy”, his roles are very robust and useful. In this case, a NodeJS role exists: <https://galaxy.ansible.com/geerlingguy/nodejs>.

You can install the role by executing “ansible-galaxy install geerlingguy.nodejs”, however, that requires you to have Ansible installed on the machine where you run the command. Since we do not have Ansible installed on your own PC, this will not work. You can either choose to install Ansible on your own PC, or you can go to the github repository of the geerlingguy.nodejs role and download it from there (<https://github.com/geerlingguy/ansible-role-nodejs/archive/refs/heads/master.zip>). You can simply plug the role in your roles/ folder and call it in your Playbook.

Notice how this role allows for the installation of NPM packages alongside NodeJS. We will need to install the NPX and Yarn packages along with NodeJS. This is what the NodeJS part of your Playbook should look like:

tasks:  
 - name: Run NodeJS role  
 include\_role:  
 name: geerlingguy.nodejs  
 vars:  
 nodejs\_npm\_global\_packages: ['npx','yarn']

Now it is time to ensure the installation of our ReactJS application. Create a separate role for this calle reactjs. Create a folder files in the reactjs role folder. Copy the react\_application folder to the files folder. Now we can have the reactjs role copy the react\_application folder to the reactserver. Use the copy module. Mind that Ansible will automatically look inside the files/ folder of the nodejs/ folder to find files when copying, so no need to explicitly mention the path. The copy module will also copy the entire directory recursively so there is no need to worry about the individual files. Mind that a trailing / does matter for the copy module. Copy folder/ will copy the **contents** of the folder to the destination. Copy folder (no trailing slash) will copy **the folder and the contents** to the destination.

Next up we want to, of course, run the React app as a service to make sure it will be started whenever we start our VM. Create a file files/etc/systemd/system/react\_application.service. Fill it with:

*[Unit]  
Description=React application*

*[Service]  
ExecStart=/usr/bin/npx serve -s -p 80 /apps/react\_application  
Restart=on-failure  
User=root  
Group=root  
Environment=PATH=/usr/bin:/usr/local/bin  
WorkingDirectory=/apps/react\_application*

*[Install]  
WantedBy=multi-user.target*

Now make sure this file is copied to /etc/systemd/system/react\_application.service. The service needs to be recognised and started as well. Use the systemd module to daemon-reload, and then use the systemd module to start the react\_application service and make sure it is enabled as well. When a service is enabled it will start automatically when the VM starts.

We are almost there but not quite yet… The React application is listening on port 80, but how will we connect to it? Port 80 is closed and worse yet, we can never reach port 80 from outside the network. Time to set up our Apache webserver to be a proxy!

First things first, we want to open up port 80 in the firewall to traffic coming from other VMs inside the network. Ansible conveniently has a firewalld module. Use it to permanently open port 80 to tcp traffic. Use the service module to make sure the firewall is restarted or else port 80 won’t be opened until you restart the VM!

Next up: set up Apache as a proxy. Inside the apache role directory, create a files/ folder, and inside of that etc/httpd/conf.d/reverseproxy.conf. Paste the following into the reverseproxy.conf:

*ProxyPass "/"* [*http://192.168.33.21:*](http://192.168.33.21:/)*80/  
 ProxyPassReverse "/"* [*http://192.168.33.21:*](http://192.168.33.21:/)*80/*

You know the drill; make sure that Ansible copies that file to /etc/httpd/conf.d/reverseproxy.conf using the copy module. Also make sure that the httpd service is restarted afterwards, or it won’t pick up on the change!

Run the Playbook again using vagrant provision ansiblehost. If all is well you should now have a fully working webproxy and React application running. You can connect to it by going to <http://127.0.0.1:8080/>.

Congratulations and I hope you learned a lot about Ansible! These are the very, very basics. There is a lot more and you can definitely optimize the structure you have now by using variables for example.