Modul: Introduction of Container

In technology, sometimes the jumps in progress are small but, as is the case with containerization, the jumps have been massive and turn the long-held practices and teachings completely upside down. With this chapter, we will take you from running a tiny service to building elastically **scalable systems** using containerization with **Docker**, the cornerstone of this revolution.

We will perform a steady but consistent ramp-up through the basic blocks with a focus on the inner workings of Docker, and, as we continue, we will try to spend a majority of the time in the world of complex deployments and their considerations.
Let’s take a look at what we will cover in this chapter :

- What are containers and why do we need them?
- Docker’s place in the container world
- Thinking with a container mindset

Bab: Automation Edit

deskripsi sub modul

Bab: Course Introduction

Bab: tugas lab automation edit

lab deskripsi

Bab: What Are Container?

In the old days, developers would develop a new application. Once that application was completed in their eyes, they would hand that application over to the operations engineers, who were then supposed to install it on the production servers and get it running. If the operations engineers were lucky, they even got a somewhat accurate document with installation instructions from the developers. So far, so good, and life was easy.

But things get a bit out of hand when, in an enterprise, there are many teams of developers that create quite different types of application, yet all of them need to be installed on the same production servers and kept running there. Usually, each application has some external dependencies, such as which framework it was built on, what libraries it uses, and so on. Sometimes, two applications use the same framework but in different versions that might or might not be compatible with each other. Our operations engineer's life became much harder over time. They had to be really creative with how they could load their ship, (their servers,) with different applications without breaking something.

Installing a new version of a certain application was now a complex project on its own, and often needed months of planning and testing. In other words, there was a lot of friction in the software supply chain. But these days, companies rely more and more on software, and the release cycles need to become shorter and shorter. We cannot afford to just release twice a year or so anymore. Applications need to be updated in a matter of weeks or days, or sometimes even multiple times per day. Companies that do not comply risk going out of business, due to the lack of agility. **So, what's the solution?**

One of the first approaches was to use **virtual machines (VMs)**. Instead of running multiple applications, all on the same server, companies would package and run a single application on each VM. With this, all the compatibility problems were gone and life seemed to be good again. Unfortunately, that happiness didn't last long. VMs are pretty heavy beasts on their own since they all contain a full-blown operating system such as Linux or Windows Server, and all that for just a single application. This is just as if you were in the transportation industry and were using a whole ship just to transport a single truckload of bananas. What a waste! That could never be profitable.

The ultimate solution to this problem was to provide something that is much more lightweight than VMs, but is also able to perfectly encapsulate the goods it needs to transport. Here, the goods are the actual application that has been written by our developers, plus – and this is important – all the external dependencies of the application, such as its framework, libraries, configurations, and more. This holy grail of a software packaging mechanism was the **Docker container**.

Developers use **Docker containers** to package their applications, frameworks, and libraries into them, and then they ship those containers to the testers or operations engineers. To testers and operations engineers, a container is just a black box. It is a standardized black box, though. All containers, no matter what application runs inside them, can be treated equally. The engineers know that, if any container runs on their servers, then any other containers should run too. And this is actually true, apart from some edge cases, which always exist.

Thus, Docker containers are a means to package applications and their dependencies in a standardized way. Docker then coined the phrase Build, ship, and run anywhere.

Bab: tes MD

`Objectives`

After completing this section, you should be able to install Ansible on a control node and describe the distinction between community Ansible and Red Hat Ansible Engine.

`Ansible or Red Hat Ansible Automation`

Red Hat provides Ansible software in special channels as a convenience to Red Hat Enterprise

Linux subscribers, and you can use these software packages normally.

However, if you want formal support for Ansible and its modules, Red Hat offers a special subscription for this, Red Hat Ansible Automation. This subscription includes support for Ansible itself, as Red Hat Ansible Engine. This adds formal technical support with SLAs and a published scope of coverage for Ansible and its core modules. More information on the scope of this support is available at Red Hat Ansible Engine Life Cycle https://access.redhat.com/support/policy/updates/ansible-engine.

CONTROL NODES

Ansible is simple to install. The Ansible software only needs to be installed on the control node (or nodes) from which Ansible will be run. Hosts that are managed by Ansible do not need to have Ansible installed. This installation involves relatively few steps and has minimal requirements.

The control node should be a Linux or UNIX system. Microsoft Windows is not supported as a control node, although Windows systems can be managed hosts.

Python 3 (version 3.5 or later) or Python 2 (version 2.7 or later) needs to be installed on the control node.

> **IMPORTANT :** If you are running Red Hat Enterprise Linux 8, Ansible 2.8 can automatically use the platform-python package that supports system utilities that use Python. You do not need to install the python36 or python27 package from AppStream.

```
[root@controlnode ~]# yum list installed platform-python
Loaded plugins: langpacks, search-disabled-repos
Installed Packages
platform-python.x86_64      3.6.8-2.el8_0     @anaconda
```

Information on how to install the ansible software package on a Red Hat Enterprise Linux system is available in the Knowledgebase article How Do I Download and Install Red Hat Ansible Engine? https://access.redhat.com/articles/3174981.

Ansible is under rapid upstream development, and therefore Red Hat Ansible Engine has a rapid life cycle. More information on the current life cycle is available at https://access.redhat.com/support/policy/updates/ansible-engine.

Red Hat provides the ansible-2.8-for-rhel-8-x86_64-rpms channel for Red Hat Enterprise Linux 8. You can also get the latest update of the Red Hat Ansible Engine 2 major release for RHEL 8 in the ansible-2-for-rhel-8-x86_64-rpms channel.

If you have a Red Hat Ansible Engine subscription, the installation procedure for Red Hat Ansible Engine 2 is as follows:

> **WARNING :** You do not need to run these steps in your classroom environment.

1. Register your system to Red Hat Subscription Manager.

```
[root@host ~]# subscription-manager register
```

2. Set a role for your system.

```
[root@host ~]# subscription-manager role --set="Red Hat Enterprise Linux Server"
```

3. Attach your Red Hat Ansible Engine subscription. This command helps you find your Red Hat Ansible Engine subscription:

```
[root@host ~]# subscription-manager list --available
```

4. Use the pool ID of the subscription to attach the pool to the system.

```
[root@host ~]# subscription-manager attach --pool=<engine-subscription-pool>
```

5. Enable the Red Hat Ansible Engine repository.

```
[root@host ~]# subscription-manager repos \
> --enable ansible-2-for-rhel-8-x86_64-rpms
```

6. Install Red Hat Ansible Engine.

```
[root@host ~]# yum install ansible
```

If you are using the version with limited support provided with your Red Hat Enterprise Linux subscription, use the following procedure:

1. Enable the Red Hat Ansible Engine repository.

```
[root@host ~]# subscription-manager refresh
[root@host ~]# subscription-manager repos --enable ansible-2-for-rhel-8-x86_64-
rpms
```

2. Install Red Hat Ansible Engine.

```
[root@host ~]# yum install ansible
```

MANAGED HOSTS

One of the benefits of Ansible is that managed hosts do not need to have a special agent installed. The Ansible control node connects to managed hosts using a standard network protocol to ensure that the systems are in the specified state.

Managed hosts might have some requirements depending on how the control node connects to them and what modules it will run on them.

Linux and UNIX managed hosts need to have Python 2 (version 2.6 or later) or Python 3 (version 3.5 or later) installed for most modules to work.

For Red Hat Enterprise Linux 8, you may be able to depend on the platform-python package. You can also enable and install the python36 application stream (or the python27 application stream).

```
[root@host ~]# yum module install python36
```

If SELinux is enabled on the managed hosts, you also need to make sure the python3-libselinux package is installed before using modules that are related to any copy, file, or template functions. (Note that if the other Python components are installed, you can use Ansible modules such as yum or package to ensure that this package is also installed.)

IMPORTANT :

1. Some package names may be different in Red Hat Enterprise Linux 7 and earlier because of the ongoing migration to Python 3.
2. For Red Hat Enterprise Linux 7 and earlier, install the python package, which provides Python 2. Instead of python3-libselinux, you will need to make sure the libselinux-python package is installed.

Some modules might have their own additional requirements. For example, the dnf module, which can be used to install packages on current Fedora systems, requires the python3-dnf package (python-dnf in RHEL 7).

NOTE :

1. Some modules do not need Python. For example, arguments passed to the Ansible raw module are run directly through the configured remote shell instead of going through the module subsystem. This can be useful for managing devices that do not have Python available or cannot have Python installed, or for bootstrapping Python onto a system that does not have it.
2. However, the raw module is difficult to use in a safely idempotent way. If you can use a normal module instead, it is generally better to avoid using raw and similar command modules. This is discussed further later in the course.

Microsoft Windows-based Managed Hosts

Ansible includes a number of modules that are specifically designed for Microsoft Windows systems. These are (listed in the Windows Modules) [https://docs.ansible.com/ansible/latest/modules/list_of_windows_modules.html] section of the Ansible module index.

Most of the modules specifically designed for Microsoft Windows managed hosts require PowerShell 3.0 or later on the managed host rather than Python. In addition, the managed hosts need to have PowerShell remoting configured. Ansible also requires at least .NET Framework 4.0 or later to be installed on Windows managed hosts.

This course uses Linux-based managed hosts in its examples, and does not go into great depth on the specific differences and adjustments needed when managing Microsoft Windows-based managed hosts. More information is available on the Ansible web site at https://docs.ansible.com/ansible/latest/user_guide/windows.html.

Managed Network Devices

You can also use Ansible automation to configure managed network devices such as routers and switches. Ansible includes a large number of modules specifically designed for this purpose. This includes support for Cisco IOS, IOS XR, and NX-OS; Juniper Junos; Arista EOS; and VyOS-based networking devices, among others.

You can write Ansible Playbooks for network devices using the same basic techniques that you use when writing playbooks for servers. Because most network devices cannot run Python, Ansible runs network modules on the control node, not on the managed hosts. Special connection methods are also used to communicate with network devices, typically using either CLI over SSH, XML over SSH, or API over HTTP(S).

This course does not cover automation of network device management in any depth. For more information on this topic, see Ansible for Network Automation [https://docs.ansible.com/ansible/latest/network/index.html] on the Ansible community website, or attend our alternative course Ansible for Network Automation (DO457) [https://www.redhat.com/en/services/training/do457-ansible-network-automation].

> **REFERENCES :** ansible(1) man page

1. Top Support Policies for Red Hat Ansible Automation
2. Installation Guide — Ansible Documentation
3. Windows Guides — Ansible Documentation
4. Ansible for Networking Automation — Ansible Documentation

Bab: Why are containers important?

These days, the time between new releases of an application become shorter and shorter, yet the software itself doesn't become any simpler. On the contrary, software projects increase in complexity. Thus, we need a way to tame the beast and simplify the software supply chain.

Also, every day, we hear that cyber-attacks are on the rise. Many well-known companies are and have been affected by security breaches. Highly sensitive customer data gets stolen during such events, such as social security numbers, credit card information, and more. But not only customer data is compromised – sensitive company secrets are stolen too.

Containers can help in many ways. First of all, Gartner found that applications running in a container are more secure than their counterparts not running in a container. Containers use Linux security primitives such as **Linux kernel namespaces** to sandbox different applications running on the same computers and **control groups (cgroups)** in order to avoid the noisy-neighbor problem, where one bad application is using all the available resources of a server and starving all other applications.

Due to the fact that container images are immutable, it is easy to have them scanned for

common vulnerabilities and exposures (CVEs), and in doing so, increase the overall security of our applications.

Another way to make our software supply chain more secure is to have our containers use a content trust. A content trust basically ensures that the author of a container image is who they pretend to be and that the consumer of the container image has a guarantee that the image has not been tampered with in transit. The latter is known as a **man-in-the-middle (MITM) attack**.

Everything I have just said is, of course, technically also possible without using containers, but since containers introduce a globally accepted standard, they make it so much easier to implement these best practices and enforce them.

OK, but security is not the only reason why containers are important. There are other reasons too.

One is the fact that containers make it easy to simulate a production-like environment, even on a developer's laptop. If we can containerize any application, then we can also containerize, say, a database such as Oracle or MS SQL Server. Now, everyone who has ever had to install an Oracle database on a computer knows that this is not the easiest thing to do, and it takes up a lot of precious space on your computer. You wouldn't want to do that to your development laptop just to test whether the application you developed really works end-to-end. With containers at hand, we can run a full-blown relational database in a container as easily as saying 1, 2, 3. And when we're done with testing, we can just stop and delete the container and the database will be gone, without leaving a trace on our computer.

Since containers are very lean compared to **VMs**, it is not uncommon to have many containers running at the same time on a developer's laptop without overwhelming the laptop.

A **third reason** why containers are important is that operators can finally concentrate on what they are really good at: provisioning the infrastructure and running and monitoring applications in production. When the applications they have to run on a production system are all containerized, then operators can start to standardize their infrastructure. Every server becomes just another Docker host. No special libraries or frameworks need to be installed on those servers, just an OS and a container runtime such as Docker.

Also, operators do not have to have intimate knowledge of the internals of applications anymore, since those applications run self-contained in containers that ought to look like black boxes to them, similar to how shipping containers look to the personnel in the transportation industry.

Bab: Example of Containerized Technology

What is Docker?

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker’s methodologies for shipping, testing, and deploying

code quickly, you can significantly reduce the delay between writing code and running it in production.

The Docker platform

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allows you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. You can easily share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Docker provides tooling and a platform to manage the lifecycle of your containers:

- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- When you’re ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

What is Podman?

Podman is a daemonless, open source, Linux native tool designed to make it easy to find, run, build, share and deploy applications using Open Containers Initiative (OCI) Containers and Container Images. Podman provides a command line interface (CLI) familiar to anyone who has used the Docker Container Engine. Most users can simply alias Docker to Podman (alias docker=podman) without any problems. Similar to other common Container Engines (Docker, CRI-O, containerd), Podman relies on an OCI compliant Container Runtime (runc, crun, runv, etc) to interface with the operating system and create the running containers. This makes the running containers created by Podman nearly indistinguishable from those created by any other common container engine.

Containers under the control of Podman can either be run by root or by a non-privileged user. Podman manages the entire container ecosystem which includes pods, containers, container images, and container volumes using the libpod library. Podman specializes in all of the commands and functions that help you to maintain and modify OCI container images, such as pulling and tagging. It allows you to create, run, and maintain those containers and container images in a production environment.

There is a RESTFul API to manage containers. We also have a remote Podman client that can interact with the RESTFul service. We currently support clients on Linux, Mac, and Windows. The RESTFul service is only supported on Linux.

What's LXC?

LXC is a userspace interface for the Linux kernel containment features. Through a powerful API and simple tools, it lets Linux users easily create and manage system or application containers.

Features

Current LXC uses the following kernel features to contain processes:

- Kernel namespaces (ipc, uts, mount, pid, network and user)
- Apparmor and SELinux profiles
- Seccomp policies
- Chroots (using pivot_root)
- Kernel capabilities
- CGroups (control groups)

LXC containers are often considered as something in the middle between a chroot and a full fledged virtual machine. The goal of LXC is to create an environment as close as possible to a standard Linux installation but without the need for a separate kernel.

Components

LXC is currently made of a few separate components:

- The liblxc library
- Several language bindings for the API:
  - python3
  - lua
  - Go
  - ruby
  - Haskell
- A set of standard tools to control the containers
- Distribution container templates

What is LXD

LXD is a next generation system container and virtual machine manager. It offers a unified user experience around full Linux systems running inside containers or virtual machines.

LXD is image based and provides images for a wide number of Linux distributions. It provides flexibility and scalability for various use cases, with support for different storage backends and network types and the option to install on hardware ranging from an individual laptop or cloud instance to a full server rack.

When using LXD, you can manage your instances (containers and VMs) with a simple command line tool, directly through the REST API or by using third-party tools and integrations. LXD implements a single REST API for both local and remote access.

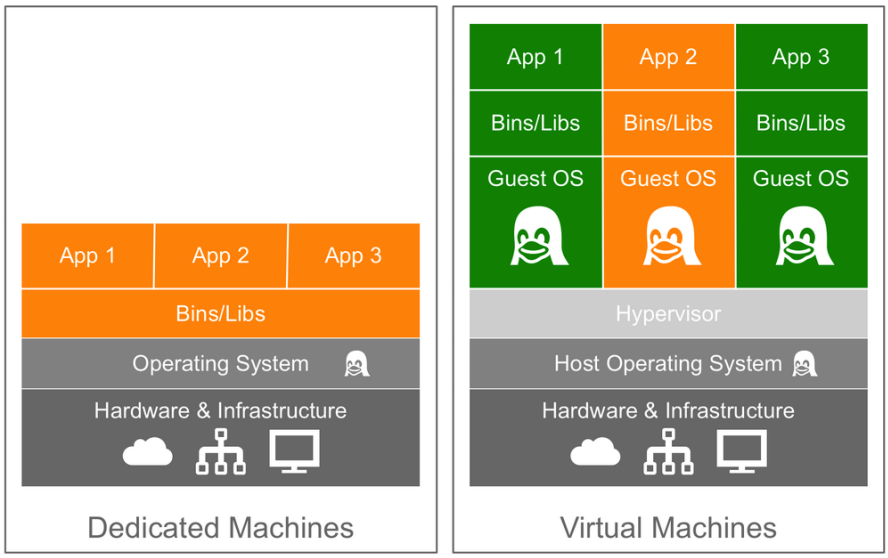
The LXD project was founded and is currently led by Canonical Ltd with contributions from a range of other companies and individual contributors.

Bab: Host, Virtual Machine and Container

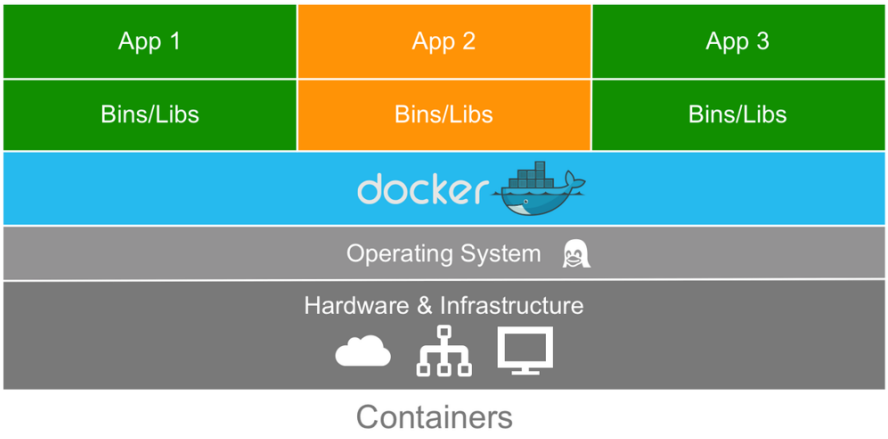
Containers are an abstracton at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly.

Virtual machines (VMs) are an abstracton of physical hardware turning one server into many servers. he hypervisor allows multple VMs to run on a single machine. Each VM includes a full copy of an operatng system, one or more apps, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

Now, let's take a look at the differences between Containers and typical virtual machine environments. The following diagram demonstrates the difference between a dedicated, bare-metal server and a server running virtual machines :



As you can see, for a dedicated machine we have three applications, all sharing the same orange software stack. Running virtual machines allow us to run three applications, running two completely different software stacks. The following diagram shows the same orange and green applications running in containers using Docker :



This diagram gives us a lot of insight into the biggest key benefit of Docker, that is, there is no need for a complete operating system every time we need to bring up a new container, which cuts down on the overall size of containers. Since almost all the versions of Linux use the standard kernel models, Docker relies on using the host operating system's Linux kernel for the operating system it was built upon, such as Red Hat, CentOS, and Ubuntu.

For this reason, you can have almost any Linux operating system as your host operating system and be able to layer other Linux-based operating systems on top of the host. Well, that is, your applications are led to believe that a full operating system is actually installed—but in reality, we only install the binaries, such as a package manager and, for example, Apache/PHP and the libraries required to get just enough of an operating system for your applications to run.

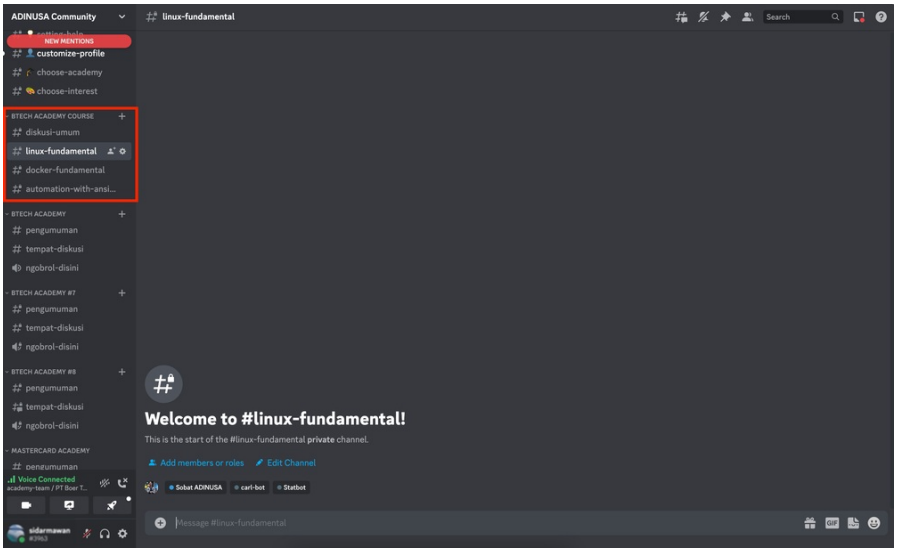
For example, in the earlier diagram, we could have Red Hat running for the orange application, and Debian running for the green application, but there would never be a need to actually install Red Hat or Debian on the host. Thus, another benefit of Docker is the size of images when they are created. They are built without the largest piece: the kernel or the operating system. This makes them incredibly small, compact, and easy to ship.

Bab: Platform Discussion

We use discord for discussions,

Join ADINUSA Discord : [ADINUSA Community]

for this course discussion on **#docker-fundamental channel**



Bab: Lab 1.1 Preparation of Lab Environment

Preparation

Before you begin, run the **nusactl login** command to login your account. Credentials for login is same with your registered credentials in this platform (ADINUSA).

```
student@pod-username-node01:~$ nusactl login
```

This guide is intended to independently setup the Docker Fundamental lab environment on a Virtual Box. This training uses 2 virtual machine with detailed specifications as follows.

In this lab environment we use the Ubuntu 22.04 OS, the image has been prepared below. For the specifications above are recommended vm specifications, you can adjust to the performance of your computer. There are 3 networks, namely NAT Network for internet access needs and Host-Only Network for remote VM needs from the Host and internal network used for communication between vm

Here's what you need to do before moving on to the next stage:

1. Download and install **VirtualBox version 7 or newer** according to the OS you are using. The latest version of **VirtualBox** can be found here

2. Download this image that contains the Ubuntu 22.04 OS and the prepared lab environment. Download **BTA-Server.ova** OneDrive

---

Setup NAT Network

1. To create a new NAT Network, click **File** menu, select the **Tools** option and select **Network Manager** Option.

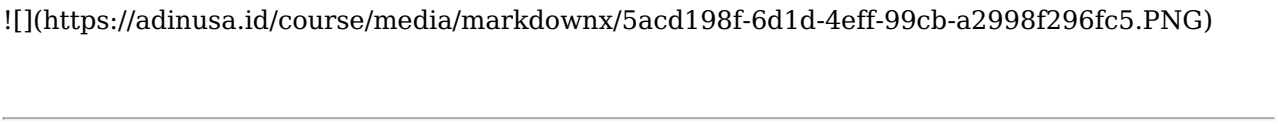
2. On the Network tab, select **NAT network** menu and select **Create**.

3. Then enter the network ip that will be used **192.168.0.0/24** and click ok.

Setup Host-Only Network

1. For Host-Only Network configuration, select **Host-Only Network** in menu.

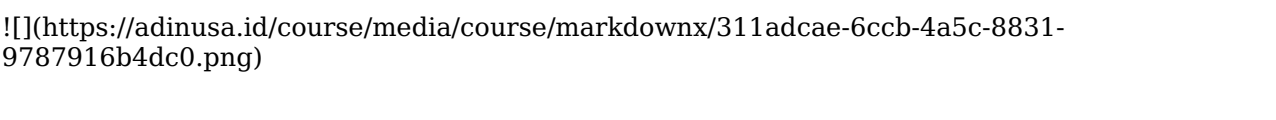
2. In **Host-Only Network Menu** on the **Adapter** tab enter IP **10.10.10.1**, netmask **255.255.255.0**, **Disable DHCP** and then **Apply**.



Import VM

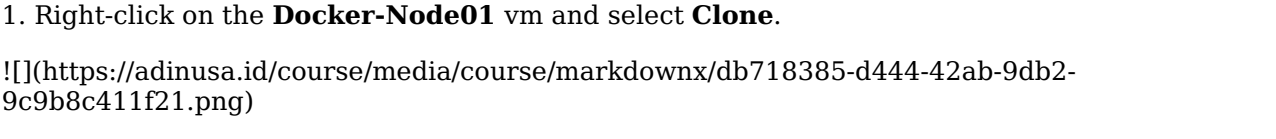
1. To import a vm, click the **File** menu and select **Import Appliance...**

2. On the import virtual Appliance tab, in the **File** column enter the ova file that was downloaded previously, then click **Next**, change the name to **Docker-Node01** then **Finish** and wait for the process.

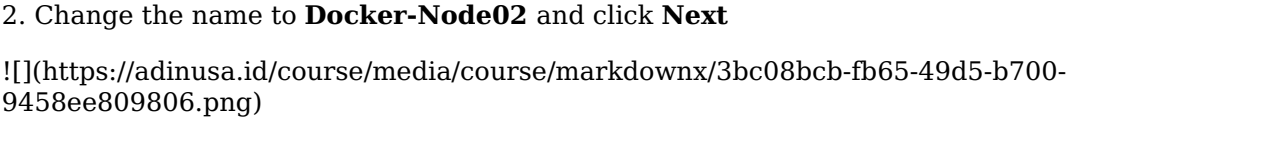


Clone VM

1. Right-click on the **Docker-Node01** vm and select **Clone**.



2. Change the name to **Docker-Node02** and click **Next**



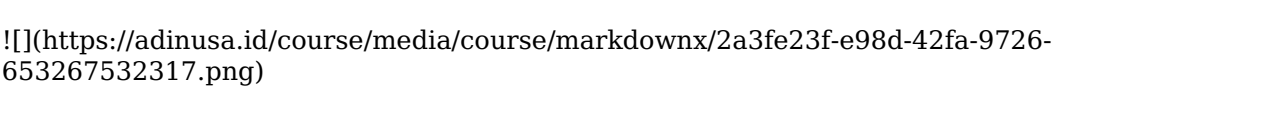
3. In Clone Type select **Full Clone** and click **Finish**

Network Configuration and Hosts Configuration on VM Docker-Node01

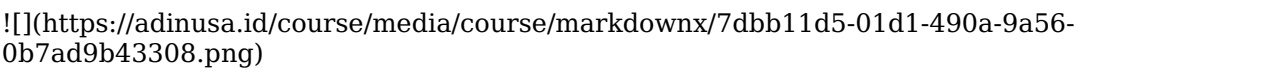
1. Turn on the vm Docker-Node01 with click start

2. The console will then open. Login with the following credentials :
user: **student**
password: **Adinusa2023**
You are free to change the password.

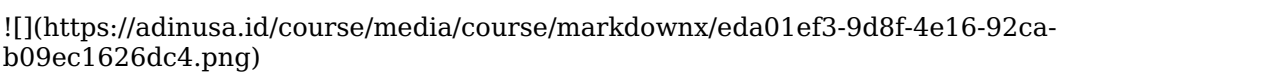
3. Edit **/etc/netplan/50-cloud-init.yaml** replace as below



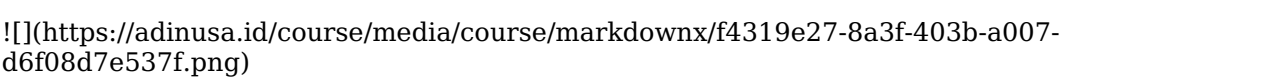
4. Use the **sudo netplan apply** command to apply the latest network configuration, then Verify with the **ip a** command.



5. Change hostname



6. Map host on /etc/hosts

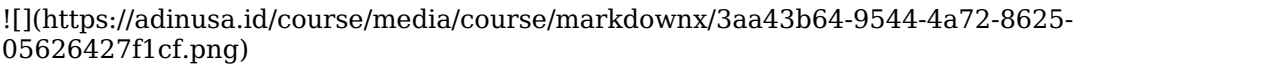


Network Configuration and Hosts Configuration on VM Docker-Node02

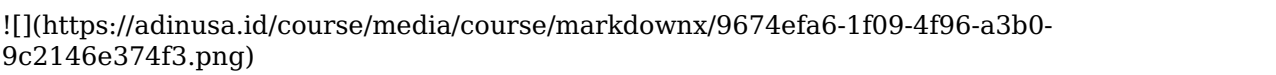
1. Turn on the vm Docker-Node02 with click start

2. The console will then open. Login with the following credentials :
user: **student**
password: **Adinusa2023**
You are free to change the password.

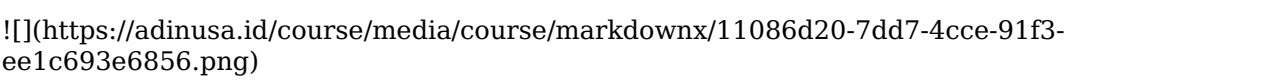
3. Edit **/etc/netplan/50-cloud-init.yaml** replace as below



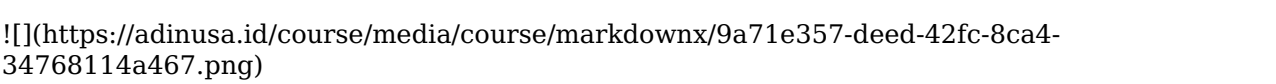
4. Use the **sudo netplan apply** command to apply the latest network configuration, then Verify with the **ip a** command.



5. Change hostname



6. Map host on /etc/hosts



Create and Distribute SSH Keygen

Execute on all nodes

4. Create a ssh-keygen so you can ssh without a password

```
student@pod-username-node01:~$ ssh-keygen -t rsa
student@pod-username-node02:~$ ssh-keygen -t rsa
```

5. Copy all public key from regular user to all nodes

```
student@pod-username-node01:~$ ssh-copy-id student@pod-username-node01
student@pod-username-node01:~$ ssh-copy-id student@pod-username-node02
...
student@pod-username-node02:~$ ssh-copy-id student@pod-username-node01
student@pod-username-node02:~$ ssh-copy-id student@pod-username-node02
```

6. Check if the host can access without using a password (passwordless).

```
student@pod-username-node01:~$ ssh student@pod-username-node01 "whoami; hostname"
student@pod-username-node01:~$ ssh student@pod-username-node02 "whoami; hostname"
...
student@pod-username-node02:~$ ssh student@pod-username-node01 "whoami; hostname"
student@pod-username-node02:~$ ssh student@pod-username-node02 "whoami; hostname"
```

Configuring nusactl requirements

Execute this config only in pod-username-node01

7. Create the /home/student/.nusactl/hosts.yaml file

```
student@pod-username-node01:~$ vim /home/student/.nusactl/hosts.yaml
...

nodes:
  pod-node1: "10.7.7.10"
  pod-node2: "10.7.7.20"
ssh-key: "/home/student/.ssh/id_rsa" # must be static path
ssh-passwd: "password_access_vm" # change the password and password must be same
every node
```

Note : Be careful this yaml file is sensitive to indents and spaces make sure it is the same as the guide.

8. Automate ssh-add automatically when logged in

```
sudo apt install -y keychain && echo "eval $(keychain -q --eval id_rsa)" >>
~/.bashrc && source ~/.bashrc
```

Note:

don't skip step number 8. if you skip it you will experience an error when grading.

> **Important Note: Make sure not to run the nusactl command in the root user**

Hints

- make sure you can ssh to all nodes without password (passwordless) from user student to user student.

- make sure you can ping all nodes using the hostname
- if you find the word **username** replace it with your adinusa account username.