# **Docker**

## **Introduction**

**Docker is an *OS virtualized software platform* that allows IT organizations to easily *create, deploy, and run applications in Docker containers,* which have all the dependencies within them.**

**The container itself is really just a very lightweight package that has all the instructions and dependencies—such as frameworks, libraries, and bins—within it.**

**The container itself can be moved from the environment to the environment very easily.**

* In a [DevOps life cycle](https://www.simplilearn.com/steps-to-building-momentum-for-devops-article), the area where Docker really shines is ***deployment***, because when you deploy your solution, you want to be able to guarantee that the code that has been tested will actually work in the production environment.
* In addition to that, when you’re building and testing the code, having a container running the solution at those stages is also beneficial because you can validate your work in the same environment used for production.

**A platform for *building, running and shipping* applications in a *consistent* manner.**

* If our application works on our development machine then it can run and function the same way on other machines.

***A situation where our application works on our development machine but doesn’t somewhere else.* This can happen for 3 reasons:**

1. **One or more files missing/ not included as part of our deployment.**
2. **Software version mismatched.**

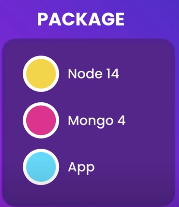
* The target machine is running a different version of some software that our application needs.
* For example: application need node version 14, however the target machine is running node version 9.

1. **Different configuration setting**

* If the configuration setting like environment variable are different across these machines.

## **Benefits**

**This is where Docker comes for the rescue,**

* ******With Docker, we can easily package up our application with everything it needs and run it anywhere on any machine with Docker.
* If our machine needs a given version of ***Node*** and ***MongoDB***, all of these will be included in our application package.
* We can run this package and run it on any machine that runs Docker.
* So if it works in development machine, it’s definitively going to work on our test and production server machines.

**If someone joins our team, they don’t have to spend half a day or so to setting up the new machine to run our application.**

* They don’t have to install and configure all these dependencies.
* They simply tell Docker to bring us up our application and Docker itself will automatically download and run these dependencies inside an isolated environment called a ***container***.

**This isolated environment (container) allows multiple applications to use different versions of some software side by side.**

* These both application can run side by side on the same machine without messing with each other.

**Keeping Development Machines Clean with Docker**

* Once we are finished with an application and no longer need to work on it, Docker allows us to easily remove the application and all of its dependencies at once.
* Without Docker, our development machines can become cluttered with numerous libraries and tools required by various applications, and it can become challenging to determine which tools can be safely removed.
* This often leads to a fear of potentially causing issues with other applications, which can make the clean-up process even more daunting.
* Docker isolated environment allows us to remove applications and their dependencies without worrying about affecting other applications, simplifying the process of cleaning up our machines.

**docker-compose down –rmi all**

# **Container vs. Virtual machines**

## **Introduction**

**A *container* is an isolated environment for running an application.**

**A Virtual machine is an abstraction of a machine (physical hardware).**

* So we can run several virtual machines on a real physical machine.
* For example, we can have a Mac and on which we can run two virtual machines like window and Linux using a tool called ***Hypervisor*.**

## **Hypervisor**

**It is software that allows multiple operating systems to run on a single physical machine by creating and managing virtual machines.**

**There are many hypervisor available like:**

* VirtualBox
* VMWare
* Hyper-V (Window only)

### **Virtual Machines: The Benefits of Isolation for Developers**

**As software developers, we can run applications in isolation inside virtual machines on the same physical machine.**

**This allows for multiple virtual machines to run completely different applications, each with its own dependencies.**

* For example, Application 1 may use Node version 14 and MongoDB version 4, while Application 2 may use Node version 9 and MongoDB version 3. Despite running on the same machine, each application runs in a separate and isolated environment.
* This is one of the key benefits of virtual machines.

### **Problems**

This model has some drawbacks.

#### **Each VM need a full-Blown OS**

**Each virtual machine requires a full copy of an operating system, which needs to be licensed, patched, and monitored.**

#### **Slow to start**

**Slow start up time, as the entire operating system needs to be loaded, similar to starting up a computer.**

#### **Resource intensive**

**Each virtual machine takes a slice of the physical hardware resources, such as CPU, memory, and disk space.**

* While we can allocate a specific amount of memory to each virtual machine, there is a limit to the number of VMs that can run on a machine due to hardware resource limitations.

## **Advantages of Containers over Virtual Machines**

### **Allow running multiple apps in isolation.**

**Containers provide the same isolation as virtual machines, allowing multiple applications to run in isolation.**

### **Lightweight**

**Unlike virtual machines, containers do not require a full operating system.**

**In fact, all containers on a single machine share the operating system of the host.**

### **Use of the host**

**This means that only a single operating system needs to be licensed, patched, and monitored.**

### **Start quickly**

**Because the operating system has already started on the host, a container can start up quickly, usually in just a second or less.**

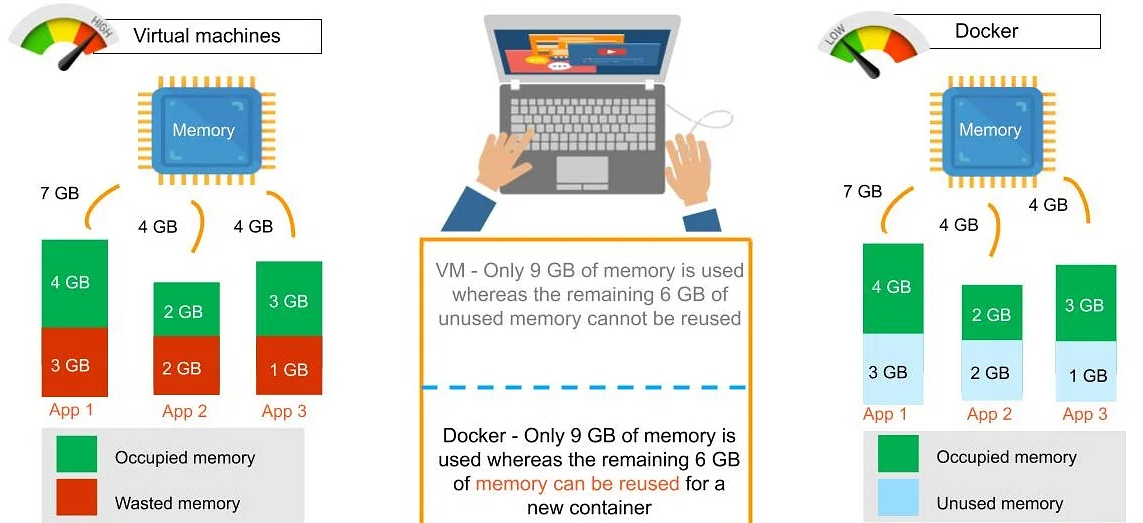
### **Need less hardware resources**

**Containers do not require a specific number of CPU cores, memory, or disk space from the host.**

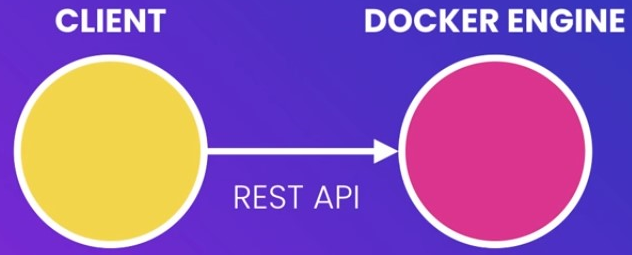
* This allows for tens or even hundreds of containers to run side-by-side on a single host.

## **Docker vs Virtual Machines**

**In the image, you’ll notice some major differences, including:**

* The virtual environment has a hypervisor layer, whereas Docker has a Docker engine layer.
* There are additional layers of libraries within the virtual machine, each of which compounds and creates very significant differences between a Docker environment and a virtual machine environment.
* With a virtual machine, *the memory usage is very high*, whereas, in a Docker environment, ***memory usage is very low.***
* In terms of ***performance***, when you start building out a virtual machine, particularly when you have more than one virtual machine on a server, the performance becomes poorer. With Docker, the performance is always high because of the single Docker engine.
* In terms of ***portability***, virtual machines just are not ideal. They’re still dependent on the host operating system, and a lot of problems can happen when you use virtual machines for portability. In contrast, *Docker was designed for portability*. You can actually build solutions in a Docker container, and the solution is guaranteed to work as you have built it no matter where it’s hosted.
* The boot-up time for a virtual machine is fairly slow in comparison to the boot-up time for a Docker environment, in which boot-up is almost instantaneous.

# **Docker Architecture**

**Docker is built on a *client-server architecture* where a client component communicates with the server component, also known as the Docker engine, using a Restful API.**

 **The Docker engine is responsible for managing the building and running of Docker containers.**

**While a container is a process similar to other processes on a computer, it is a special type of process.**

**Unlike virtual machines, containers do not include a full operating system; instead, they share the kernel of the host operating system.**

**The *kernel* is the core of the operating system and manages applications and hardware resources such as memory and CPU.**

* As different operating systems have their own kernels with different APIs, applications developed for one OS cannot run on another OS's kernel.
* For instance, Windows applications cannot run on Linux as they need to communicate with the Windows kernel.

However, with the custom-built Linux kernel that Windows 10 now includes, both Windows and Linux containers can run on Windows.

* On a Mac, the kernel is different from Linux and Windows kernels and does not have native support for running containers continuously.
* Therefore, Docker on Mac uses a lightweight Linux virtual machine to run Linux containers.

## **Components of Docker**

**There are four components:**

1. **Docker client and server**
2. **Docker image**
3. **Docker registry**
4. **Docker container**

### **Docker client and server**

**This is a command-line-instructed solution in which you would use the terminal on your Mac or Linux system to issue commands from the Docker client to the Docker daemon.**

**The communication between the Docker client and the Docker host is via a REST API.**

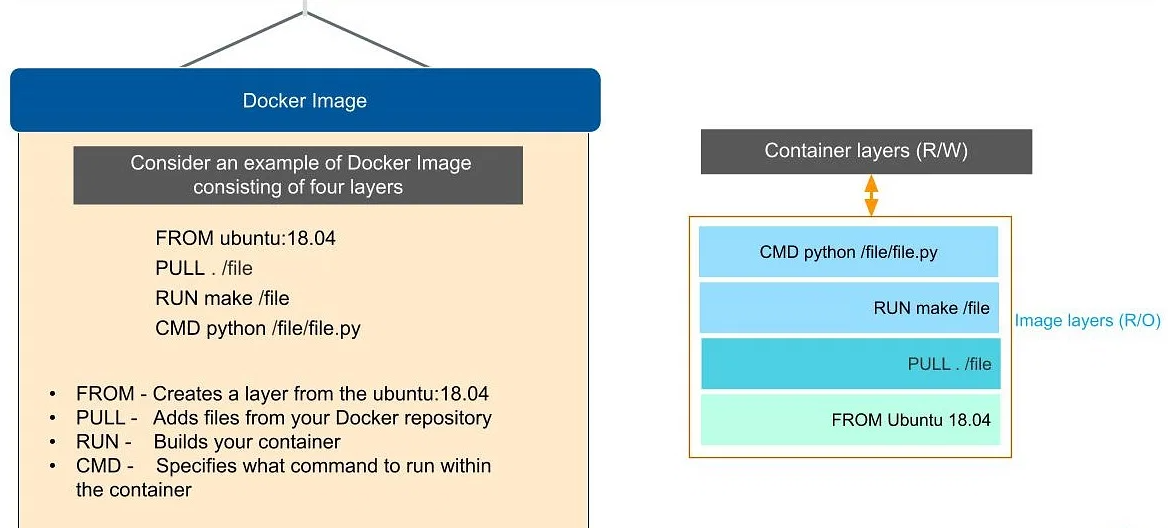
* You can issue similar commands, such as a Docker Pull command, which would send an instruction to the daemon and perform the operation by interacting with other components (image, container, registry).
* The Docker daemon itself is actually a server that interacts with the operating system and performs services.
* As you’d imagine, the Docker daemon constantly listens across the REST API to see if it needs to perform any specific requests.
* If you want to trigger and start the whole process, you’ll need to use the Docker command within your Docker daemon, which will start all of your performances.
* Then you have a Docker host, which lets you run the Docker daemon and registry.

### **Docker Image**

**A Docker image is a template that contains instructions for the Docker container.**

* That template is written in a language called YAML, which stands for Yet Another Markup Language.

**The Docker image is built within the YAML file and then hosted as a file in the Docker registry.**

* The image has several key layers, and each layer depends on the layer below it.
* Image layers are created by executing each command in the ***Dockerfile*** and are in the read-only format.
* You start with your base layer, which will typically have your base image and your base operating system, and then you will have a layer of dependencies above that.
* These then comprise the instructions in a read-only file that would become your Dockerfile.

**Here we have four layers of instructions:**

1. ***From***: command creates a layer based on Ubuntu, and then we add files from the Docker repository to the base command of that base layer.
2. ***Pull***: Adds files from your Docker repository
3. ***Run***: Builds your container
4. ***CMD***: Specifies which command to run within the container

**In this instance, the command is to run Python.**

* One of the things that will happen as we set up multiple containers is that each new container adds a new layer with new images within the Docker environment.
* Each container is completely separate from the other containers within the Docker environment, so you can create your own separate read-write instructions within each layer.
* What’s interesting is that ***if you delete a layer***, *the layer above it will also get deleted*.

**What happens when you pull in a layer but something changed in the core image?**

* Interestingly, the main image itself cannot be modified.
* Once you’ve copied the image, you ***can modify*** it locally.
* *We can* ***never*** *modify the actual base image.*

# **Install Docker**

**Check current version of Docker in system in PowerShell/Command line.**

**~ docker version**

**Client:**

**Cloud integration: v1.0.31**

**Version: 23.0.5**

**API version: 1.42**

**Go version: go1.19.8**

**Git commit: bc4487a**

**Built: Wed Apr 26 16:20:14 2023**

**OS/Arch: windows/amd64**

**Context: default**

**Server:**

**Docker Desktop 4.19.0 (106363)**

**Engine:**

**Version: 23.0.5**

**API version: 1.42 (minimum version 1.12)**

**Go version: go1.19.8**

**Git commit: 94d3ad6**

**Built: Wed Apr 26 16:17:45 2023**

**OS/Arch: linux/amd64**

**Experimental: false**

**containerd:**

**Version: 1.6.20**

**GitCommit: 2806fc1057397dbaeefbea0e4e17bddfbd388f38**

**runc:**

**Version: 1.1.5**

**GitCommit: v1.1.5-0-gf19387a**

**docker-init:**

**Version: 0.19.0**

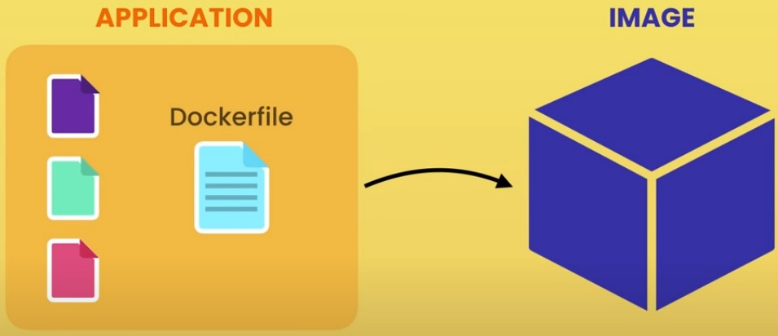
**GitCommit: de40ad0**

## **WSL 2 backend Requirement**

* **Windows 11 64-bit: Home or Pro version 21H2 or higher, or Enterprise or Education version 21H2 or higher.**
* **Windows 10 64-bit: Home or Pro 21H1 (build 19043) or higher, or Enterprise or Education 20H2 (build 19042) or higher.**
* **Enable the WSL 2 feature on Windows. For detailed instructions, refer to the Microsoft documentation.**
* **The following hardware prerequisites are required to successfully run WSL 2 on Windows 10 or Windows 11:**
* *64-bit processor with Second Level Address Translation (SLAT)*
* *4GB system RAM*
* *BIOS-level hardware virtualization support must be enabled in the BIOS settings*.
* *Download and install the Linux kernel update package.*

# **Development Workflow**

**We can dockerize any type of application, regardless of how it's built.**

**Dockerizing an application involves making a small change by adding a docker file to it.**

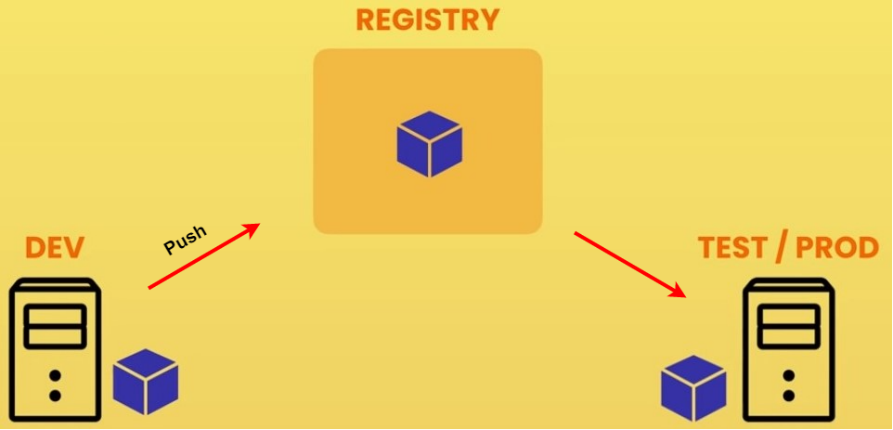
**The docker file contains instructions that Docker uses to package the application into an image that includes everything it needs to run, such as**

* ***A cut-down operating system,***
* ***a runtime environment like Node or Python,***
* ***application files,***
* ***third-party libraries,***
* ***environment variables, etc.***

We create a docker file and give it to docker for packaging our application into an image.

**Once we have the image, we can use it to start a container and run the application inside an isolated environment.**

* This is much simpler than launching the application directly and running it inside a typical process.



Furthermore, we can push the image to a ***Docker registry*** like **Docker Hub**, which is a storage for Docker images that anyone can use.

* Once the image is on Docker Hub, we can start the application on any machine running Docker by telling Docker to start a container using the image.
* We no longer need to maintain complex release documents; all the instructions for building the application image are written in the docker file.
* With Docker, we can package up our application into an image and run it virtually anywhere.

# **Docker in Action**

## **Open Command Line/ PowerShell.**

**//make new directory**

**mkdir hello-docker**

**// open the new directory**

**cd hello-docker**

**// open the visual studio code**

**code .**

## **Create a JavaScript file as sample project.**

**We want to use Docker to build, run, and ship our application, instead of having to install Node on another computer to run it.**

### **Instructions to run a app.js**

* **Start with an OS**
* **Install Node**
* **Copy app files**
* **Run node app.js**

## **Add a docker file name with “Dockerfile” with no extension.**

Install the docker extension in the visual studio code.

**By writing instructions in a Dockerfile, we can package up our application and its dependencies, avoiding the need for a complex release document.**

**// choose base image**

**From node:alpine**

**// copy the application or application files**

**Copy . /app**

**// set the current working directory**

**WORKDIR /app**

**// to execute command – to run the application on node.**

**CMD node app.js**

**We start by choosing a base image, such as a Linux image or a Node image, and copy our application files into it.**

* Then, we use the command instruction to execute a command, such as running our app.js file.
* These images are published officially on ***Docker Hub***.
* We can visit ***hub.docker.com*** and search for ***'node'*** to see the official Node image.
* Docker Hub serves as a registry for Docker images.

## **Open the terminal in the visual Studio code to run commands**

### **Build the image**

**We can then tell Docker to build our image and give it a tag name using the command:**

**docker build -t hello-docker .**

* **-t: used to give a tag to our image**
* **. (full stop): used to represent current directory**

**In VS Code, you might expect to find an image file in the current directory, but there is actually nothing there.**

* Docker does not store an image as a single file, and the process for how it stores images is complex, but we don't need to worry about that.
* To see all the images on your computer, you can type "***docker images***" or "***docker image ls***" (short for list) in the terminal.
* On this machine, there is a repository called "hello docker" that contains an image with the "latest" tag, which was added by Docker by default.

**In this repository, there is an image with the "latest" tag, which Docker added by default.**

* We use these tags for versioning our images.
* This allows each image to contain a different version of our application.

### **Run the image**

**After building our image, we can run it on any computer with Docker installed using the command:**

**docker run hello-docker**

**We can run this image on any computer that has Docker installed. On my development machine, I can use the command *docker run* followed by the image name "*hello-docker*".**

* It doesn't matter which directory I am currently in because this image contains all the necessary files to run our application.

### **Publish the image to DockerHub**

**To publish our image to Docker Hub, we can use the "docker push" command.**

docker push

**As a user, I can publish my Docker image to Docker Hub, allowing others to use it on their machines.**

* This enables me to easily run the image on a test or production machine by pulling and running it.
* To demonstrate this, I will search for "*Play with Docker*" and log in with my Docker ID.
* I can then start a new virtual machine, which will have only a Linux operating system and Docker installed.
* If I try to run the "node" command, I will receive an error saying "node command not found", since Node.js is not installed on the virtual machine.

### **Pull the image in other system.**

**//docker pull <dockerId>/<Image-Name>**

**docker pull shubhamk/hello-docker**

**By adding a Dockerfile to an application, we can containerize it, which involves creating an image that includes instructions for packaging the application.**

* After creating the image, we can run it on any machine with Docker, allowing us to deploy the application virtually anywhere.

# **Linux Command Line**

**Docker is built on basic Linux concepts, so knowing Linux commands is important for troubleshooting and productivity in Docker.**

## **Linux Distribution**

**A Linux distribution (often abbreviated as "*distro*") is an operating system made from a collection of software based on the *Linux kernel*.**

* These are ***specialized versions*** of ***open source Linux software*** created by individuals and communities to meet specific needs, such as running servers, desktop computers, and mobile phones.
* It typically includes a ***package management system***, ***software repositories***, and ***pre-configured system settings***.
* Different distributions can have different goals, such as being optimized for a specific use case or user interface.
* Some popular examples of Linux distributions include Ubuntu, Debian, CentOS, and Fedora.
* While *most distributions support the same set of commands, there may be some differences, so it's important to be aware of them.*

## **Running Linux**

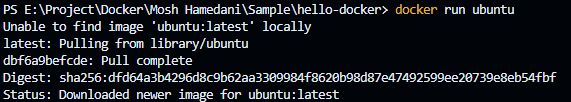
### **Pull Ubuntu image to run in local host.**

**docker pull Ubuntu**

**OR**

**docker run Ubuntu**

**I am going to execute the command "*docker run ubuntu*".**

* If the image is already present in the local system, Docker will create a container using that image.
* Otherwise, Docker will first download the image from Docker Hub and then create the container.

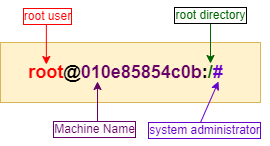
**To interact with a container and begin its execution, we need to use the command "docker run -it" where the flags "-it" stand for interactive mode.**

**docker run -it Ubuntu**

**The "root" part in the command prompt represents the currently logged-in user, which is by default the root user with the highest privileges.**

**The name of the machine is represented after the "@" symbol.**

* **The forward slash "/" represents the root directory.**

**The "#" symbol indicates that the user has the highest privileges, while the "$" symbol represents a normal user.**

**Linux uses a forward slash to separate files and directories, which is different from Windows that uses a backslash.**

* Additionally, Linux is case-sensitive, so uppercase and lowercase letters are different.
* We can use the up and down arrows to go through the executed commands and use the history command to see all the executed commands.
* We can also replay any of the commands using an exclamation mark followed by the command number.

**When we enter the command "echo $0", the output displays the location of the shell program that is currently running.**

* The output shows that the program is located in the ***"/bin/bash"*** directory.
* The term "bash" is actually an acronym for "***Bourne-Again SHell***," named after Steve Bourne, who created the original shell program.
* The "***bash***" program is an *improved version of the original shell program* with additional features and capabilities.

## **Package Manager**

**apt is a powerful package manager in Ubuntu and other Debian-based distributions.**

* It has an abbreviation for Advanced Package Tool.
* We can search for, install, update, and remove packages on your system.
* We can also use apt with other sub-commands to perform more advanced operations, such as ***adding new repositories***, ***pinning packages to specific versions***, or ***fixing broken dependencies***.

|  |  |
| --- | --- |
| Command | Description |
| apt update | **Updates the local package index with the latest information from the repositories.** |
| apt upgrade | **Upgrades all installed packages to their latest available versions.** |
| apt install <package> | **Installs a new package or updates an existing package.** |
| apt remove <package> | **Removes a package from the system.** |
| apt search <package> | **Searches for a package by name or keyword.** |
| apt show <package> | **Displays detailed information about a package, including its dependencies, size, and version.** |
| apt list | **Lists all installed packages.** |

***It is recommended to run 'apt update' before installing any package to ensure that the package database is updated.***

* *Once the update is complete, you can proceed with installing the desired package.*

# **Commands**

|  |  |
| --- | --- |
| Command | Description |
| docker version | **Echoes Client’s and Server’s Version of Docker** |
| docker images | **List all Docker images** |
| docker build <image> | **Builds an image form a Docker file** |
| docker save <path> <image> | **Saves Docker image to .tar file specified by path** |
| docker run   <image-name> | **Runs a command in a new container.** |
| docker start | **Starts one or more stopped containers** |
| docker stop <container\_id> | **Stops container** |
| docker rmi <docker\_image\_id> | **Removes Docker image** |
| docker rm <container\_id> | **Removes Container** |
| docker pull | **Pulls an image or a repository from a registry** |
| docker push | **Pushes an image or a repository to a registry** |
| docker export | **Exports a container’s filesystem as a tar archive** |
| docker exec | **Runs a command in a run-time container** |
| docker ps | **Show running containers** |
| docker ps -a | **Show all containers** |
| docker ps -l | **Show latest created container** |
| docker search | **Searches the Docker Hub for images** |
| docker attach | **Attaches to a running container** |
| docker commit | **Creates a new image from a container’s changes** |

**There are two ways to do that:**

* **Using docker ps command (older and popular method)**
* **Using docker container command (newer and less known method)**

|  |  |  |
| --- | --- | --- |
| ps command | container command | Description |
| docker ps | **docker container ls** | **Show running docker containers** |
| docker ps -a | **docker container ls -a** | **List all docker containers** |
| docker ps -f "status=exited" | **docker container ls -f "status=exited"** | **Display only stopped containers** |
| docker ps -q | **docker container ls -q** | **Display only container id** |
| docker ps -l | **docker container ls -l** | **Display latest created container** |
|  | **docker container ls -a --filter "ancestor=image\_name"** | **Show containers associated with an image** |
|  |  |  |

## **From Command**

**In a Dockerfile, the FROM instruction establishes a new build stage and defines the Base Image for future instructions.**

* Therefore, it's essential to begin a valid Dockerfile with a FROM instruction.
* We can choose any valid image to use as the base, and it's often convenient to start with a pre-existing image from the Public Repositories.

### **The syntax for the `FROM` instruction is as follows:**

**FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]**

**OR**

**FROM [--platform=<platform>] <image> [AS <name>]**

**OR**

**FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]**

* <***image***>: is the name of the base image to use.
* <***tag***>: is an optional tag for the image. If no tag is specified, the `latest` tag will be used by default.
* ***AS <name>:*** is an optional alias for the image.

"***ARG***" is the only instruction that can come before "***FROM***" in the Dockerfile.

#### **Here's an example `FROM` instruction in a Dockerfile:**

**// FROM node:<node-version>-<linux distribution>**

**FROM node:14-alpine**

**This specifies that the base image for the new Docker image will be the `*node*` image with the `14-alpine` tag.**

We can also specify multiple `FROM` instructions in a Dockerfile to create multiple images or use one build stage as a dependency for another.

* Each `FROM` instruction clears any state created by previous instructions.
* Additionally, you can use the optional `AS` keyword to name a new build stage.
* This name can be used in subsequent `FROM` and `COPY` --from=<name> instructions to refer to the image built in this stage.

#### **Here's an example Dockerfile that uses multiple `FROM` instructions:**

**FROM node:14-alpine AS builder**

**WORKDIR /app**

**COPY . .**

**RUN npm install**

**RUN npm run build**

**FROM nginx:latest**

**COPY --from=builder /app/dist /usr/share/nginx/html**

**The first `FROM` instruction specifies the `*node:14-alpine*` image as the base image and names it `*builder*`.**

* The subsequent `COPY`, `RUN`, and `WORKDIR` instructions are executed in this build stage.
* node:<version>-alpine

**The second `FROM` instruction specifies the `nginx:latest` image as the base image.**

* The ‘**COPY --from=builder**’ instruction copies the output of the previous build stage to the ***‘/usr/share/nginx/html’*** directory in the new image.