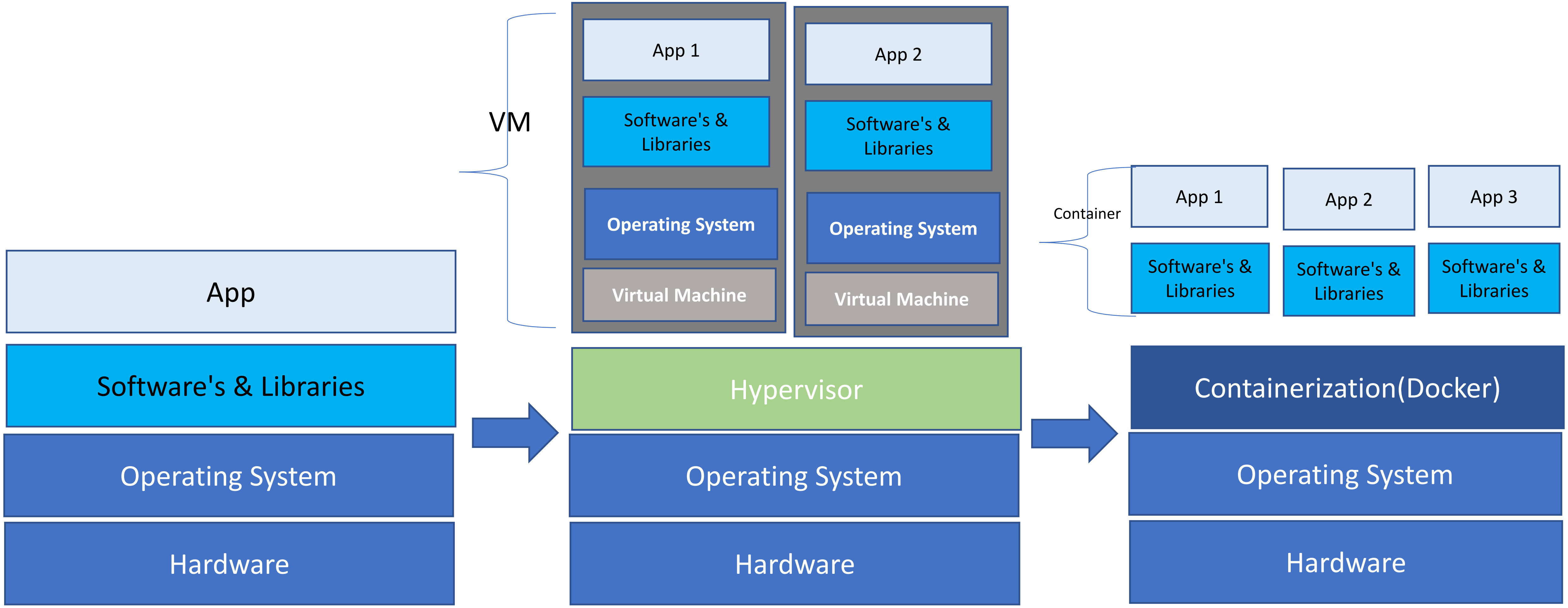
Mithun Technologies
devopstrainingblr@gmail.com

Program Agenda

- Introduction
- Virtualization and Containerization
- Virtual Machine and Docker
- Installation
- Dockerfile, Docker Image and Docker Container
- Docker Commands
- Build the Dockerfile
- Create Docker Custom images and Containers, push to Docker Hub
- Docker Compose
- Docker Swarm

Past & Present Of Application Deployment Approach



Traditional Deployment

Virtualized Deployment

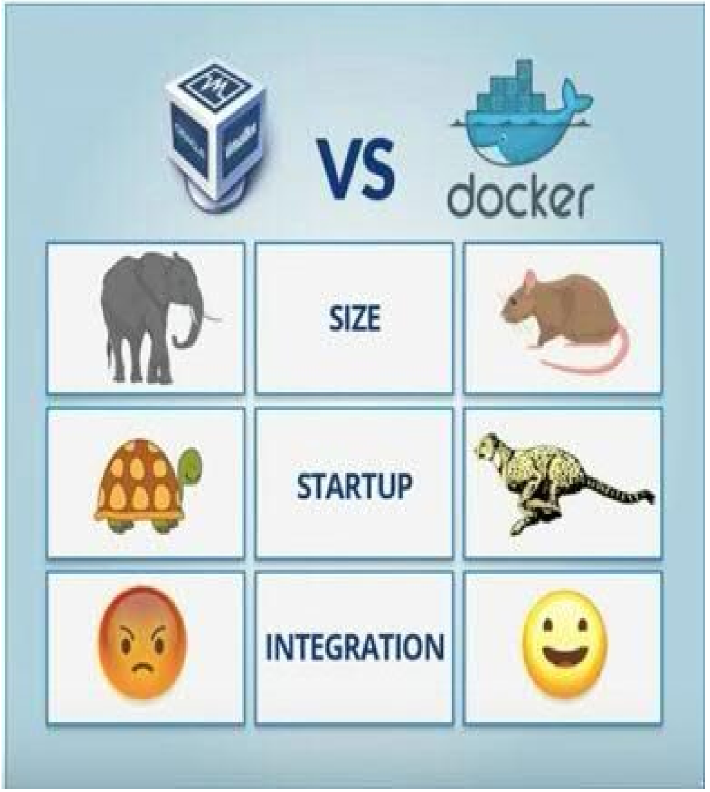
- Requires guest OS in every VM.
- Needs a Hypervisor application to create virtual machines.
- VMs are isolated from rest of the system
- Uses the resources like CPU, memory and storage from host machine.

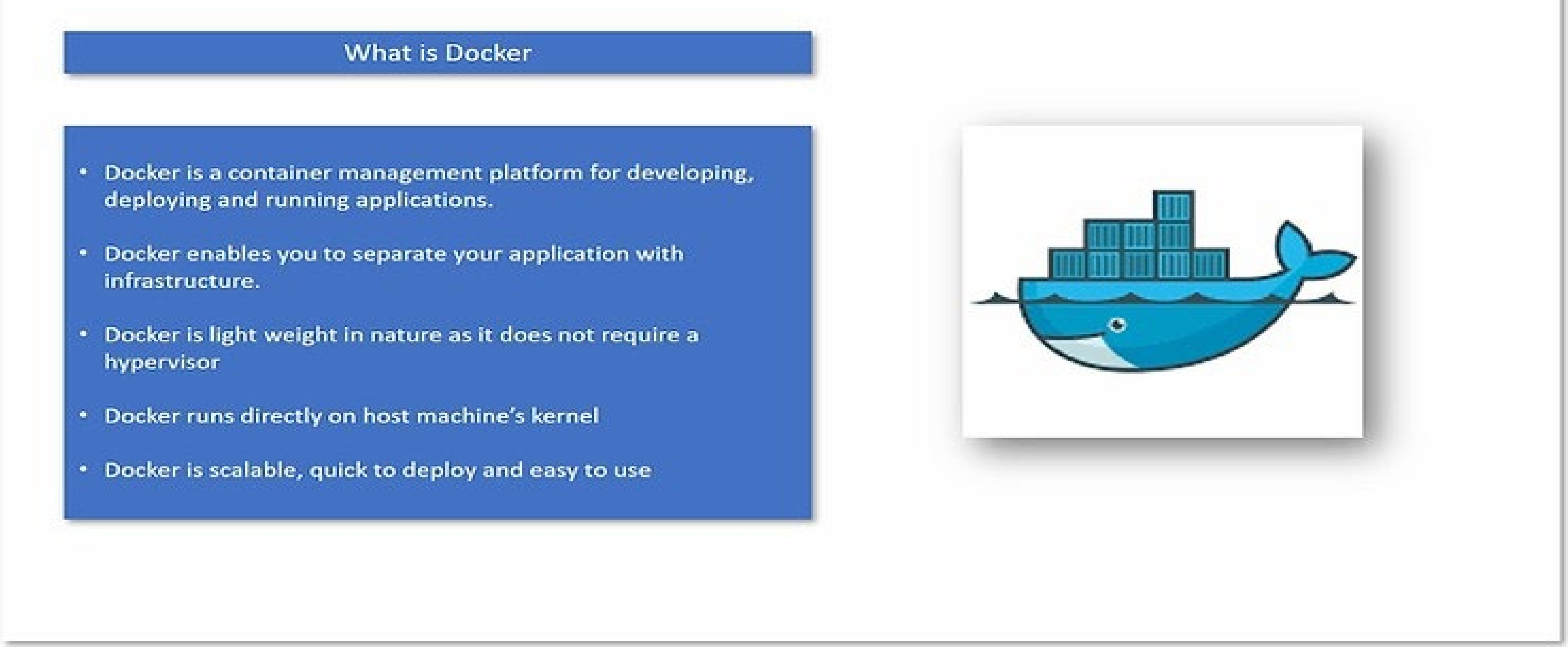
Container Deployment

- Does not require guest OS as it shares host OS between containers.
- Does not require a Hypervisor.
- Docker containers packages are self-contained and can run applications in any environment.

Virtual Machines Vs Containers

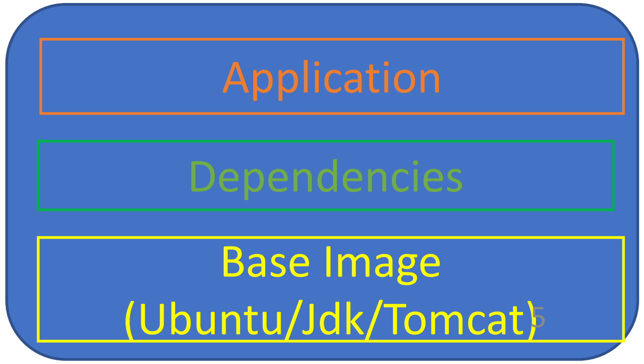
Virtual Machine	*Container*
• Heavyweight	• Lightweight
• Limited performance	• Native performance
• Each VM runs in its own OS	• All containers share the host OS
• Hardware-level virtualization	• OS virtualization
• Startup time in minutes	• Startup time in milliseconds
• Allocates required memory	• Requires less memory space
• Fully isolated and hence more secure	• Process-level isolation, possibly less secure





- ❖ *Docker is a containerisation platform which packages your app and its all dependencies together in the form of containers. so as to ensure that your application works seamlessly in any environment be it dev or test or prod.*
- ❖ *Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications*
- ❖ *Docker is available in two editions: Community Edition (CE) and Enterprise Edition (EE).*

EX: Base Image + Configurations & Dependencies (Software's) + Application



Why Containers?

Developers care because:

- Quickly create ready-to-run packaged applications, low cost deployment and replay
- Automate testing, integration, packaging
- Reduce / eliminate platform compatibility issues (“It works in dev!”)
- Support next gen applications (microservices)

IT cares because:

- Improve **speed** and frequency of releases, reliability of deployments
- Makes app lifecycle efficient, consistent and repeatable – configure once, run many times
- Eliminate environment inconsistencies between development, test, production
- Improve production application resiliency and scale out / in on demand

Type	Containerization Tool
Vendor	Docker
Is Open Source?	Yes – Some extent
Operating system	Cross Platform
URL	https://www.docker.com/

<u>Docker Installation</u>

Docker For Linux

Docker For Windows
Windows 10 64-bit Home/Pro Only.
Hyper V Should be enabled.

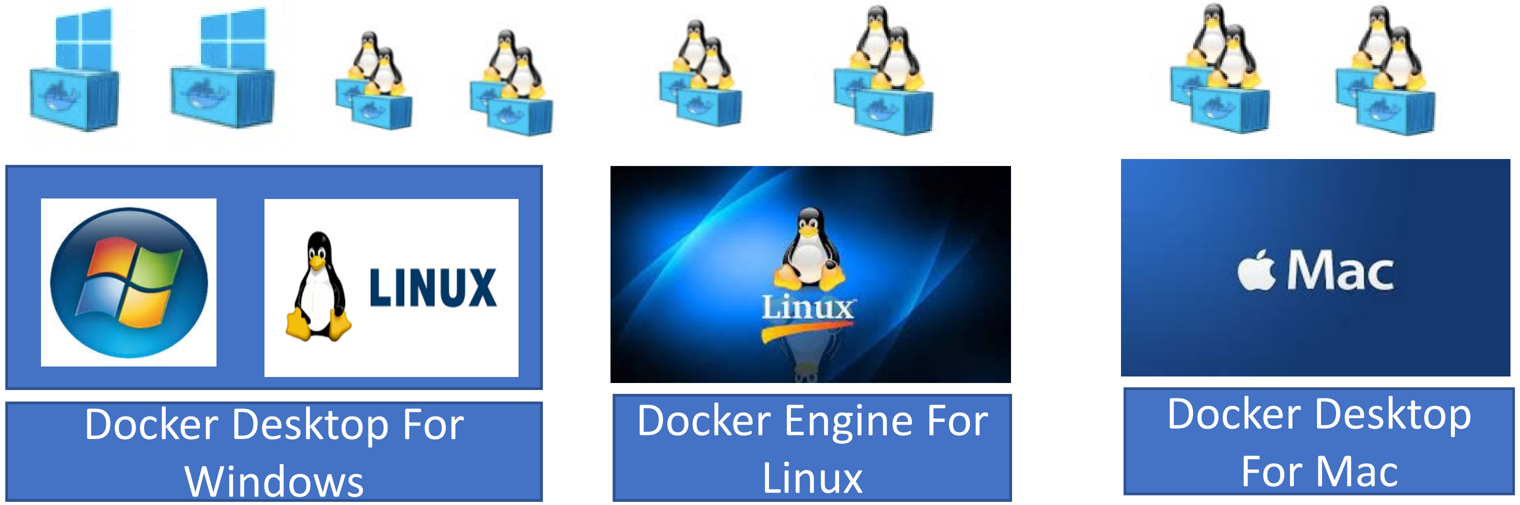
Docker For Mac

https://docs.docker.com/engine/install/

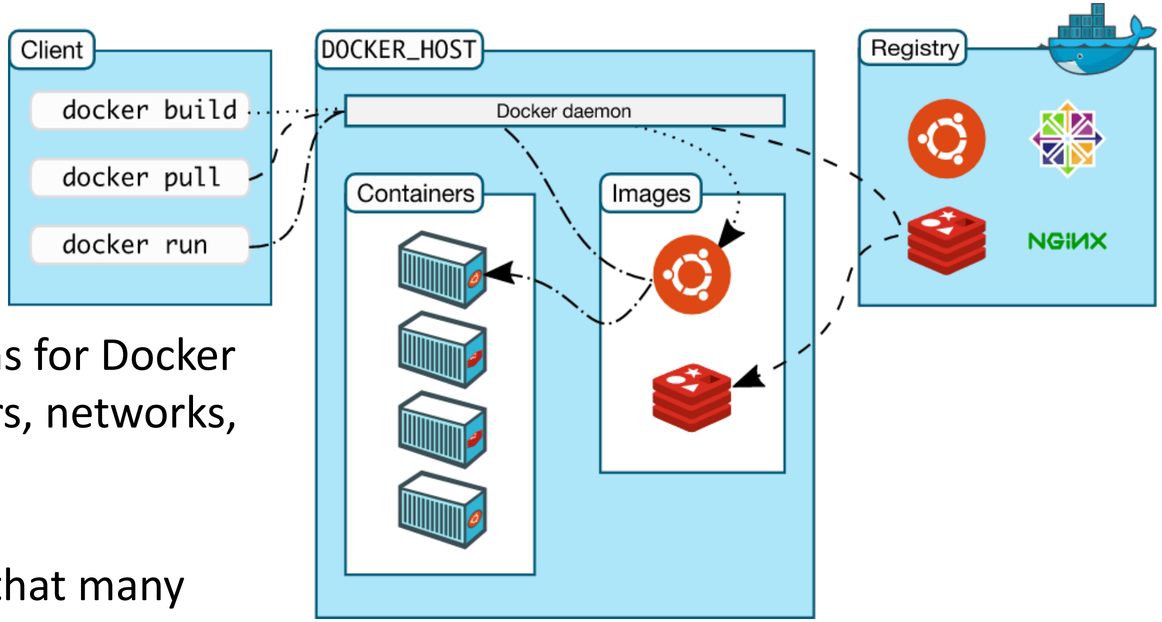
Docker Hub Registry:

Register in https://hub.docker.com/

Note: Docker Desktop Installed in the Windows Machine can run Linux Based Docker Containers and Windows-based Docker Containers.
But You cannot run Windows Based Docker Containers on Docker Engine installed in Linux.

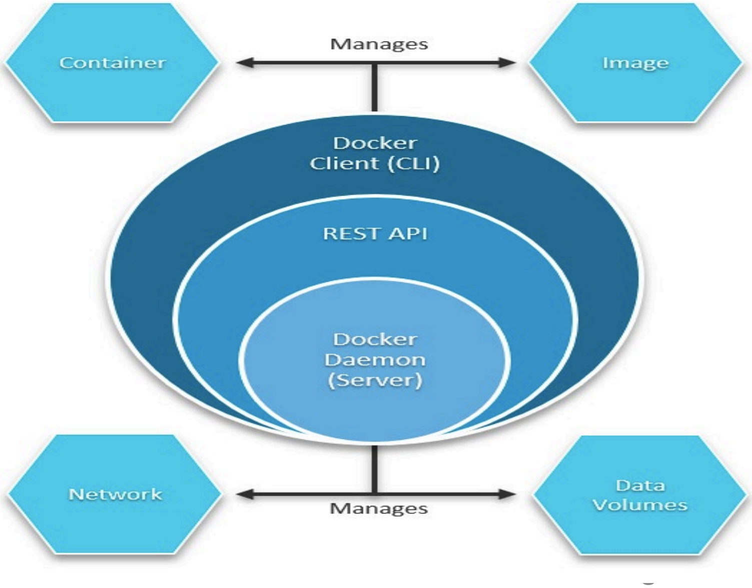


Architecture of Docker showing how it works



Docker daemon: The Docker daemon (dockerd) is process that listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. Etc.

Docker client(CLI): The Docker client (docker cli) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd . The docker cli uses the Docker API internally.

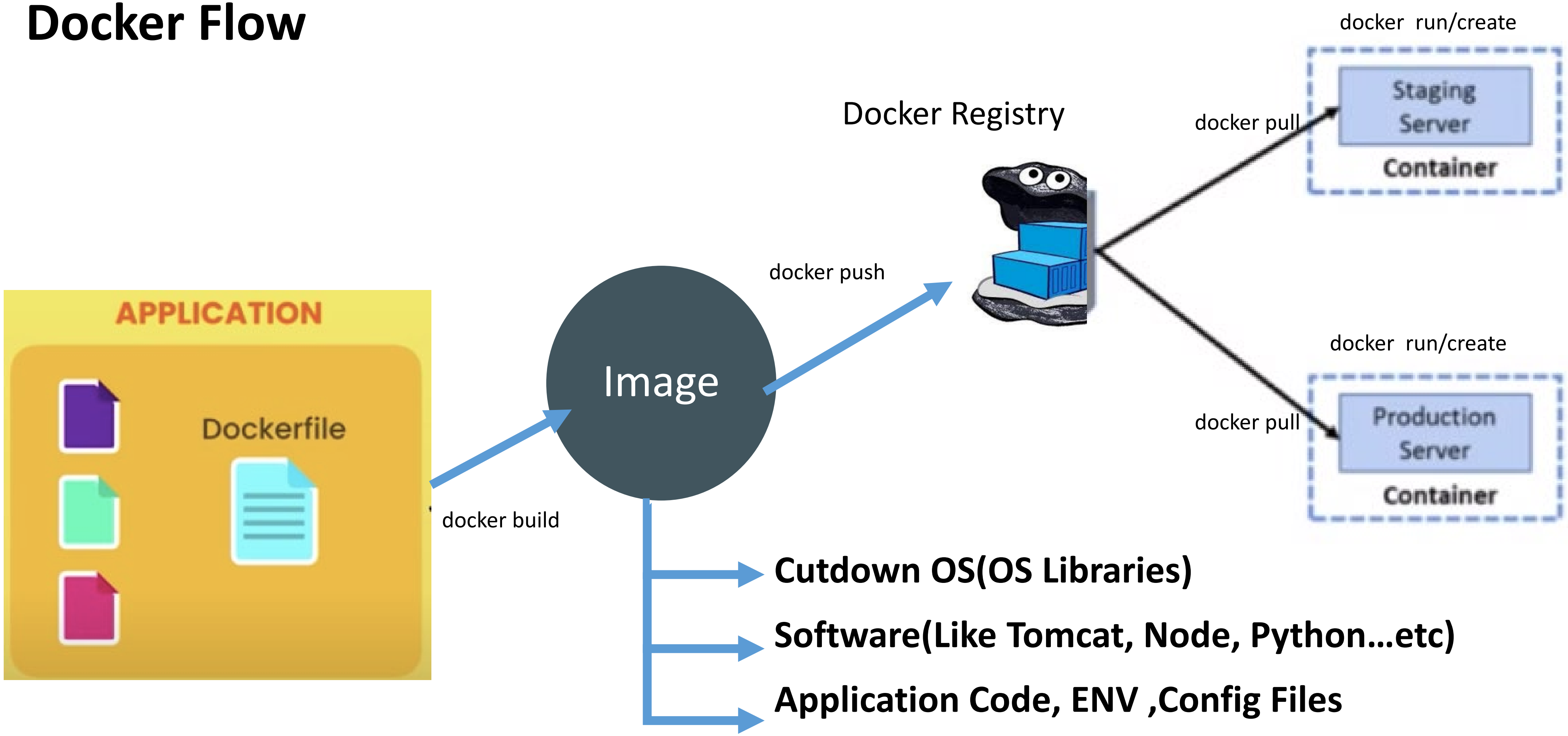


Docker registries:

A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

Docker Flow



APPLICATION
Dockerfile
docker build
Image
docker push
Docker Registry
docker pull
docker run/create
Staging Server
Container
docker pull
docker run/create
Production Server
Container
Cutdown OS(OS Libraries)
Software(Like Tomcat, Node, Python…etc)
Application Code, ENV ,Config Files

Docker Objects

When you use Docker, you are creating and managing images, containers, networks, volumes, and other objects.
This section is a brief overview of some of those objects.

Docker Images: An image is a package with instructions for creating a Docker container , that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the tomcat image, With your application code, as well as the configuration details needed to make your application run.

To build your own image, you create a *Dockerfile* with a simple syntax for defining the steps needed to create the image and run it.
Each instruction in a Dockerfile creates a layer in the image.

Docker Containers: A container is a runtime instance of an image. You can create, start, stop, move, or delete a container using the Docker CLI or API. You can connect a container to one or more networks, attach storage(volumes) to it.

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(server).

Docker Networks: Networking is about communication among processes, and Docker's networking is no different. Docker networking is primarily used to establish communication between Docker containers and the outside world via the host machine where the Docker daemon is running.

Docker Volumes: By default, all files created inside a container are stored on a writable container layer and the data doesn't persist when that container no longer exists. Docker has two options for containers to store files in the host machine or network/cloud storage, so that the files are persisted even after the container stops: **Bind Mounts**, and **Volumes**

Advantages of Docker

- ***Rapid application deployment*** *– containers include the minimal run time requirements of the application, reducing their size and allowing them to be deployed quickly.*

- ***Portability across machines*** *– an application and all its dependencies can be bundled into a single container that is independent from the host version of Linux kernel, platform distribution, or deployment model. This container can be transferred to another machine that runs Docker using it's docker image, and executed there without compatibility issues.*

- ***Version control and component reuse*** *– you can track successive versions of a images, roll-back to previous versions. Containers reuse components from the preceding layers(base images), which makes them noticeably lightweight.*

- ***Sharing*** *– you can use a remote repository to share your image with others. And it is also possible to configure your own private repository.*

- ***Lightweight footprint and minimal overhead*** *– Docker images are typically very small, which facilitates rapid delivery and reduces the time to deploy new application containers. Allowing for more efficient use of computing resources, both in terms of energy consumption and cost effectiveness.*

- ***Simplified maintenance*** *– Docker reduces effort and risk of problems with application dependencies. Multiple containerized apps on a single server don't mess up each other. If you update an app, you just build a new image, run fresh containers and don't have to worry about other ones on the machine breaking*

Advantages of Docker

- ***Self-sufficient***

  A Docker container contains everything it needs to run

  Minimal Base OS

  Libraries and frameworks

  Application code

  A Docker container should be able to run anywhere that Docker can run.

- ***Docker shines for microservices architecture.***

  NOTE: Containers aren’t required to implement microservices, but they are perfectly suited to the microservices approach and to agile development processes generally.

Build & Deployment Process

To be more clear,
In 1990'S we had physical servers where apps are hosted and served.

In 2000'S we had Virtual Machines where apps are hosted and served ,running on top of the Physical servers. Virtualization started here.

In early 2013 the concept of Containerization picks up where we can package the application and its dependencies so that whatever a developer released will work regardless of which environment it runs under(DEV, QA & PROD).

Before Docker
Build

- A developer pushes a change to SCM tools.
- The CI sees that new code is available. It rebuilds the project, runs the tests and generates a application package(like jar/war/ear ..etc.)
- The CI saves application package as build artifacts into repositories.

Deployment

- Setup dependencies(Software's) and configuration files in App(Deployment) Servers(Physical/Virtual Machines).
- Download the application package
- Start the application

Build & Deployment Process

Cons:

- Packaging and deploying applications in multiple environments is a real and challenging job.
- Setup Environments before deploying an application.
- Applications may work in one environment which may not work in another environment. The reasons for this are varied; different operating system, different dependencies, different libraries, software versions.

After Docker

Build

- A developer pushes a change to SCM tools.
- The CI sees that new code is available it. It rebuilds the project, runs the tests and generates an application package(jar/war/ear ..etc) & and also we will create docker image(application code(jar/ear/war..etc), dependencies (Software's) and configuration files).
- CI Will push docker image to the docker repo(Public Repo(Docker Hub) or Private Repo(Nexus/JFrog/DTR. Etc.)).

Deployment

- Download the Docker image(package).
- Start the docker image which will create a container where our application will be running.

Advantages:

Application works seamlessly in any environment be it dev or test or prod.

Dockerfile

Docker file is a simple text file that consists of instructions to build(assemble) Docker image. Docker can build images by reading the instructions from a Docker file. Docker runs instructions in a Docker file in order. There are a variety of **Dockerfile instructions** we can put in our Docker file. These include **FROM ,COPY,ADD,RUN,CMD,ENTRYPOINT, ARG,ENV, LABEL, WORKDIR, USER, EXPOSE** and **VOLUME. Etc**.

FROM

- The FROM instruction initializes a new build stage and sets the *Base Image* for subsequent instructions.
- FROM must be the first non-comment instruction in the Docker file. As such, a valid Docker file must start with a FROM instruction.
- ARG is the only instruction that may precede FROM in the Docker file.
- FROM can appear multiple times within a single Docker file to create multiple images or use one build stage as a dependency for another.
- Optionally a name can be given to a new build stage by adding AS name to the FROM instruction. The name can be used in subsequent FROM and COPY --from=<name|index> instructions to refer to the image built in this stage.
- The tag or digest values are optional. If you omit either of them, the builder assumes a latest tag by default. The builder returns an error if it cannot find the tag value.

Usage

FROM <image>
FROM <image>:<tag>
FROM <image>@<digest

Dockerfile

MAINTAINER

- The MAINTAINER instruction allows you to set the Author field of the generated images.

Usage

MAINTAINER <name>

COPY

- Copies files or directories(local files/directories part of build context) from <src> and adds them to the filesystem of the image at the path <dest>.
- <src> must be relative to the source directory that is being built (the context of the build).
- <dest> is an absolute path, or a path relative to WORKDIR.
- If <dest> doesn't exist, it is created along with all missing directories in its path.

Usage

COPY <src> <dest>

COPY ["<src>", "<dest>"] (this form is required for paths containing whitespace)

ADD

- The ADD instruction copies files, directories (local files/directories part of build context) or remote files (HTTP(S) Endpoints) from <src> and adds them to the filesystem of the image at the path <dest>.
- At first glance you may notice that COPY and ADD seems to perform the same operations. However, ADD also supports 2 additional feature's. First, you can use a URL to add files from remote sources. Secondly, If you add tar file using ADD it will extract a tar file from the source directly into the destination.

Dockerfile

- A valid use case for ADD is when you want to extract a local tar file into a specific directory in your Docker image.
- If you’re copying in local files to your Docker image, always use COPY because it’s more explicit.

Usage

ADD <src> <dest>
ADD ["<src>", "<dest>"] (this form is required for paths containing whitespace)

RUN

- The RUN instruction will execute any commands in a new layer on top of the current image(previous layer) and commit the results.
- The resulting committed image will be used for the next step in the Docker file.
- RUN instructions will be executed while building the image.
- We can use RUN instructions to install & configure required software's and confutations in image while building image.
- We can have multiple RUN instructions in a Docker file. All RUN instructions will be executed by docker while building image.

Usage

RUN <command> <param1> <param2> (shell form, the command is run in a shell, which by default is /bin/sh -c)
RUN ["<executable>", "<param1>", "<param2>"] (exec form)

Normal shell processing does not occur when using the exec form. For example, RUN ["echo", "$HOME"] will not do variable substitution on $HOME

Dockerfile

CMD

- CMD instruction specifies the command to run when a container is launched(started).
- CMD instruction will be executed while starting the container.
- The main purpose of a CMD is to provide defaults(command or process to be executed) while starting the container.
- There can only be one CMD instruction in a Docker file. If you list more than one CMD in a Docker file then only the last CMD will take effect.
- CMD instruction can be overridden at run time . If the user specifies arguments to docker run <Image> <args> then they will override the default specified in CMD.

Usage

CMD ["<executable>","<param1>","<param2>"] (exec form, this is the preferred form)
CMD <command> <param1> <param2> (shell form)
CMD ["<param1>","<param2>"] (as default parameters to ENTRYPOINT)

Normal shell processing does not occur when using the exec form. For example, CMD ["echo", "$HOME"] will not do variable substitution on $HOME.

ENTRYPOINT

- An ENTRYPOINT allows you to configure a container that will run as an executable.
- ENTRYPOINT instruction will be executed while starting the container.
- ENTRYPOINT looks similar to CMD, because it also allows you to specify a command with parameters.
- The difference is ENTRYPOINT command and parameters are not ignored(can’t override).

Dockerfile

- We can have ENTRYPOINT & CMD together in a Docker file in such case CMD will be not executed as a command or executable CMD parameters will be used as default parameters for ENTRYPOINT.
- If the user specifies arguments to docker run <Image> <args> then they will be appended after all elements in an exec form of ENTRYPOINT. Will override all elements specified using CMD.
- If you list more than one ENTRYPOINT in a Docker file Only the last ENTRYPOINT instruction in the Docker file will have an effect.

Usage

ENTRYPOINT ["<executable>", "<param1>", "<param2>"] (exec form, preferred)
ENTRYPOINT <command> <param1> <param2> (shell form)

ARG

- Defines a variable that users can pass at build-time to the builder with the docker build command using the --build-arg <varname>=<value> flag.
- Multiple variables may be defined by specifying ARG multiple times.
- ARG is the only instruction that may precede FROM in the Docker file.
- ARG values are not available after the image is built. A running container won’t have access to an ARG variable value.

Usage

ARG <name>[=<default value>]

Dockerfile

ENV

- The ENV instruction sets the environment variable <key> to the value <value>.
- This value will be in the environment(available/accessible) for all subsequent instructions in the build stage Docker file commands and can be replaced inline as well.
- The environment variables set using ENV will persist when a container is run from the resulting image.

Usage

```
ENV <key> <value>
ENV <key>=<value> [<key>=<value> ...]
```

LABEL

- The LABEL Docker file instruction adds metadata to an image. A LABEL is a key-value pair.
- To include spaces within a LABEL value, use quotes and backslashes as you would in command-line parsing.
- Labels are additive including LABELs in FROM(base) images.
- If Docker encounters a label/key that already exists, the new value overrides any previous labels with identical keys.
- To view an image’s labels, use the docker inspect command. They will be under the "Labels" JSON attribute.

Usage

```
LABEL <key>=<value>
LABEL <key>=<value> [<key>=<value> ...]
```

Dockerfile

WORKDIR

- The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Docker file.
- If the WORKDIR doesn’t exist, it will be created even if it’s not used in any subsequent Docker file instruction.
- The WORKDIR instruction can be used multiple times in a Docker file.

Usage

WORKDIR </path/to/workdir>

USER

- The USER instruction sets the username or UID to use when running the image and for any RUN, CMD and ENTRYPOINT instructions that follow it in the Docker file.
- It’s recommend to start container process as non root user as security best practice.

Usage

USER <username | UID>

Dockerfile

EXPOSE

- The EXPOSE instruction tells Docker that the container listens on the specified network ports at runtime. Default is TCP if the protocol is not specified.
- The EXPOSE instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published. To actually publish the port when running the container, use the -p flag on docker run to publish and map one or more ports.

Usage

EXPOSE <port>
EXPOSE <port>/<protocol>

VOLUME

- The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host(Unnamed volumes). Each time you create a new container it will create a new volume and use it.
- There is a different way to use persistent volumes while creating containers using –v or –mount options.

Usage

VOLUME <path>
VOLUME <path> <path>

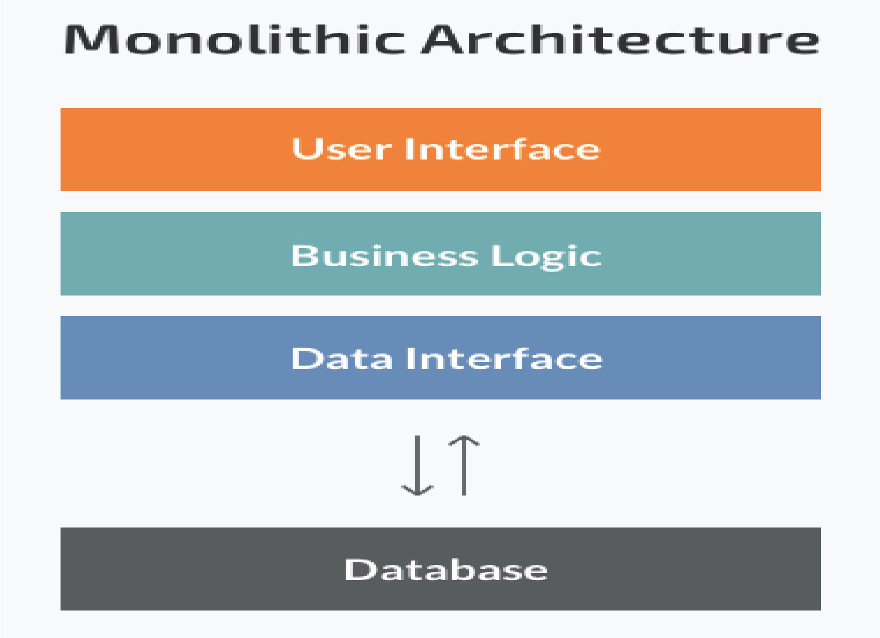
Monolithic VS Microservices Architecture

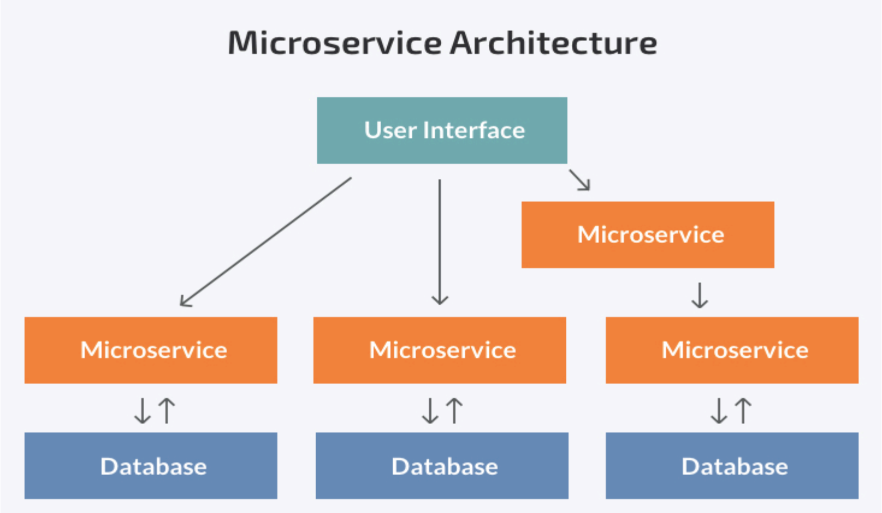
Monolithic Architecture

Monolithic application has single code base with multiple modules. Modules are divided as either for business features or technical features. It has single build system which build entire application and/or dependency. It also has single executable or deployable binary.

Microservices Architecture

Microservices are a software development technique that arranges an application as a collection of loosely coupled services. Which can be developed, deployed, and maintained independently. Each of these services is responsible for discrete task and can communicate with other services through simple APIs to solve a larger complex business problem.





Monolithic

Pros

- Simple to develop
- Simple to deploy single package(war/jar/ear,..etc.).
- Easy to debug & Error tracing.
- Simple to test.

Cons

- Difficult to understand and modify.
- Tightly coupled.
- Redeploy entire app on each update.
- Single bug can bring down entire application.
- Scaling the application is difficult. If we need to scale only few features/modules will end up scaling entire app as its single package.
- Changes on one section(module/features of the code can cause
- impact on the other section of the code as it's a single code base.

Micro Services

Pros

- Loosely coupled.
- Easy to understand & modify as it's small code base.
- Better deployments as each service(feature/module) can be deployed independently.
- Each service can be scaled independently as each service is a separate package.
- Each service can developed using any new technology as each service is a separate code base(Repository).

Cons

- **Communication between services is complex**: Since everything is now an independent service, you have to carefully handle requests traveling between your modules/services using Rest API's.
- **Integration testing is difficult**: Testing a microservices-based application can be cumbersome. In a monolithic approach, we would just need to launch our WAR on an application server and ensure its connectivity with the underlying database. With microservices, each dependent service needs to be deployed before testing can occur.
- **Debugging problems can be harder**: Each service has its own set of logs to go through. Log, logs, and more logs.
- **Deployment challengers**: The Application may need coordination among multiple services, which may not be as straightforward as deploying a WAR in a container.

Questions ?

Thank You