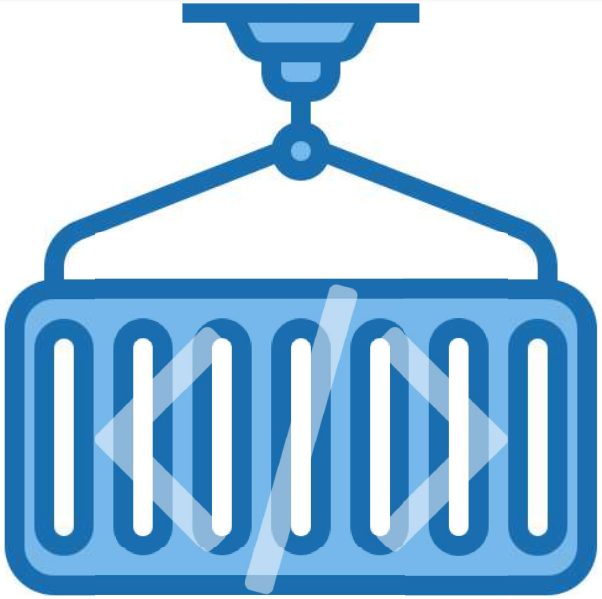
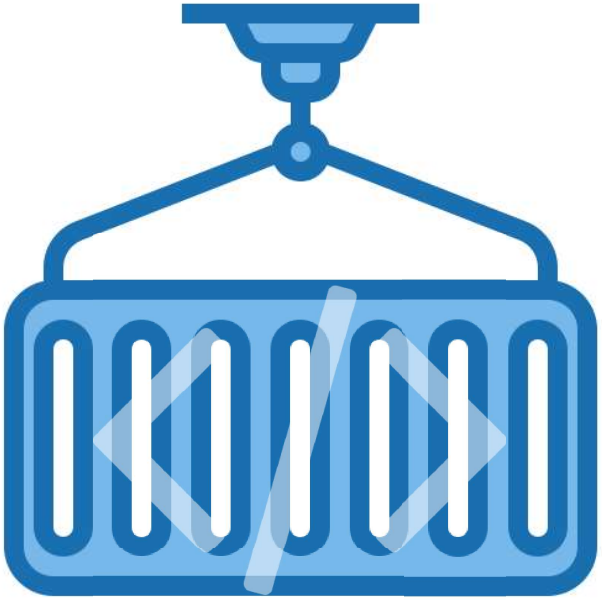
Docker Container Exec

Docker enables users to execute commands or interact with the container shell directly. This is done using the `docker exec` command



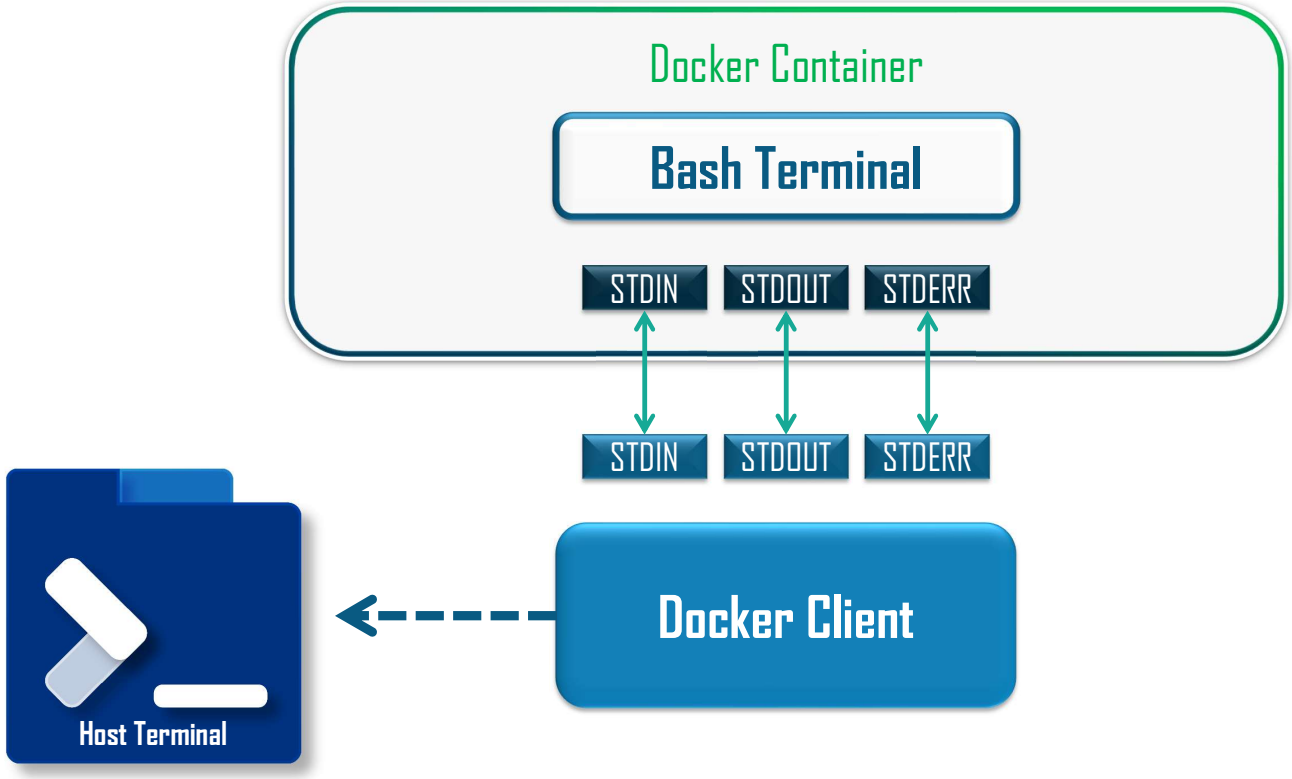
Docker Container Exec

- The exec command in its default mode executes a single command on the container
- The more popular approach is to launch a bash terminal inside the container. This is done using the `bash` option
- To interact with the container terminal, the `-it` option must be passed
- To execute the commands as a root user `-u` flag followed by `0` is used



edureka!

Docker Exec Command Execution




```
docker exec -it <containerName> /bin/bash
```

Dockerfile Reference

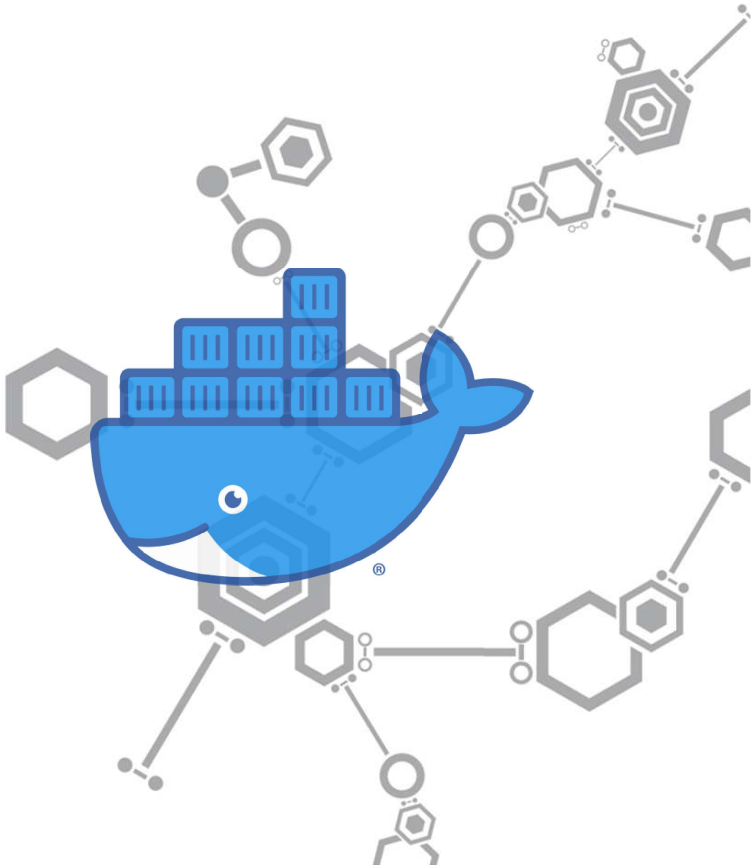
- Docker builds images by taking instructions from a file called the Dockerfile
- The Dockerfile is a text document with commands to assemble an image
- The `docker build` command executes all the instructions in sequence to create the image

Dockerfile: Working

- The build command refers to the Dockerfile and a context
- The context is a set of files specified at the PATH or URL location
- PATH is used for local files and URL for is used for remote repositories

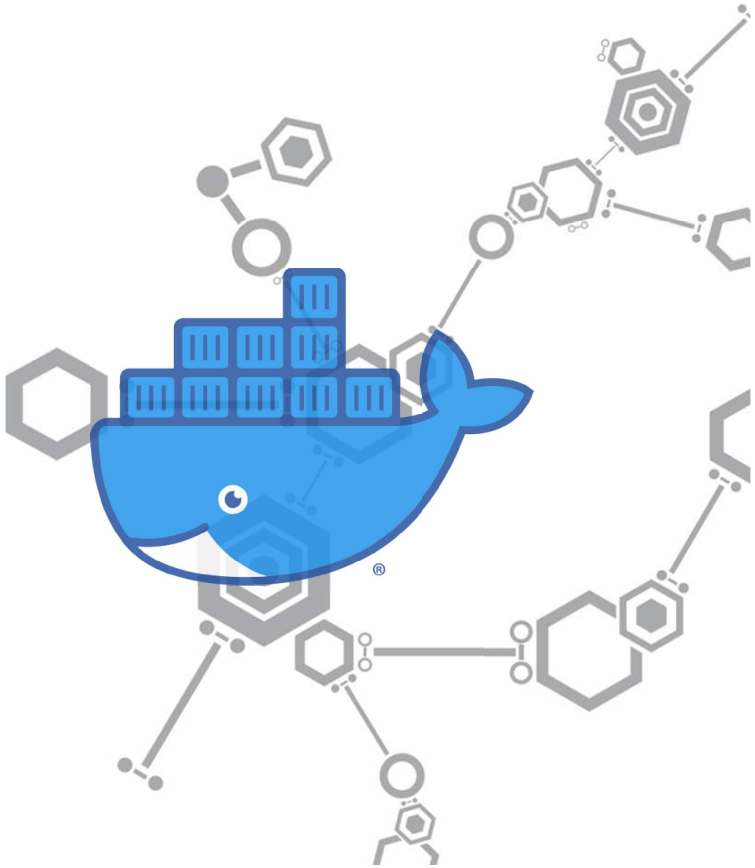
Dockerfile Instruction

- The docker image is made of read-only layers
- Each of the layers corresponds to an instruction in the Dockerfile
- These layers, when stacked together, represent an image
- Executing the image and generating a container adds a writable layer on top



Dockerfile Instruction: Example

```
FROM ubuntu:18.04
RUN  apt install -y apache2
COPY index.html /var/www/html/
CMD  ["/usr/sbin/httpd", "-D", "FOREGROUND"]
EXPOSE 80
```



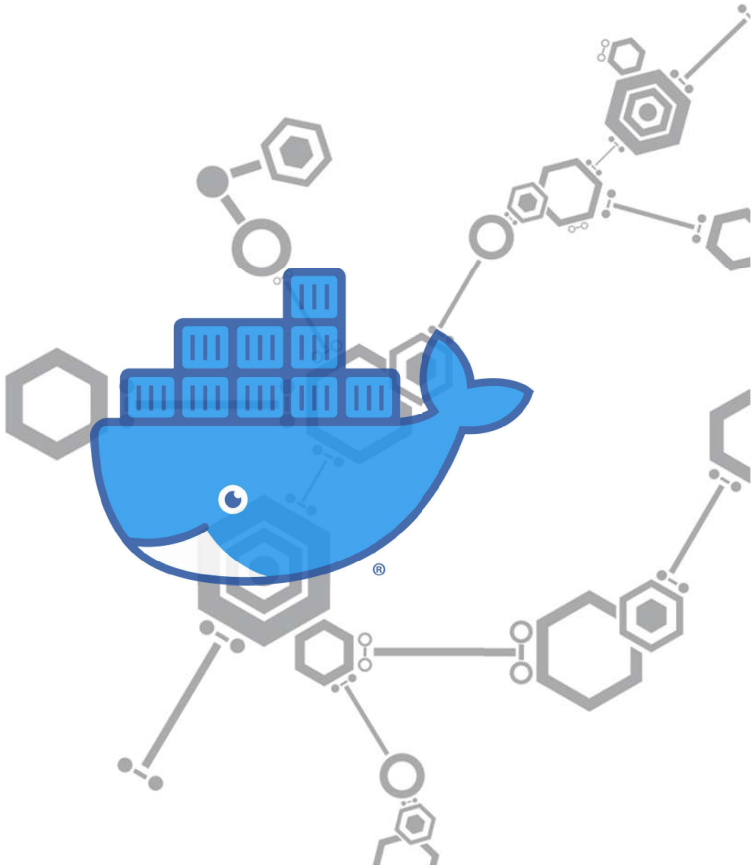
edureka!

Dockerfile: The Instruction and its Layer

Commands	Description
FROM	Creates the bottom most layer of ubuntu 18.04 for the image
RUN	Installs apache httpd server on top of the ubuntu layer
COPY	Copies files from the local directory to the container
CMD	Specifies the command to run when the container is live
EXPOSE	Exposes the container port to the system

Dockerfile: Build Context

While issuing the build command for a Dockerfile, the current working directory is considered as the build context. The Docker daemon collects all the files from the build context to build the image

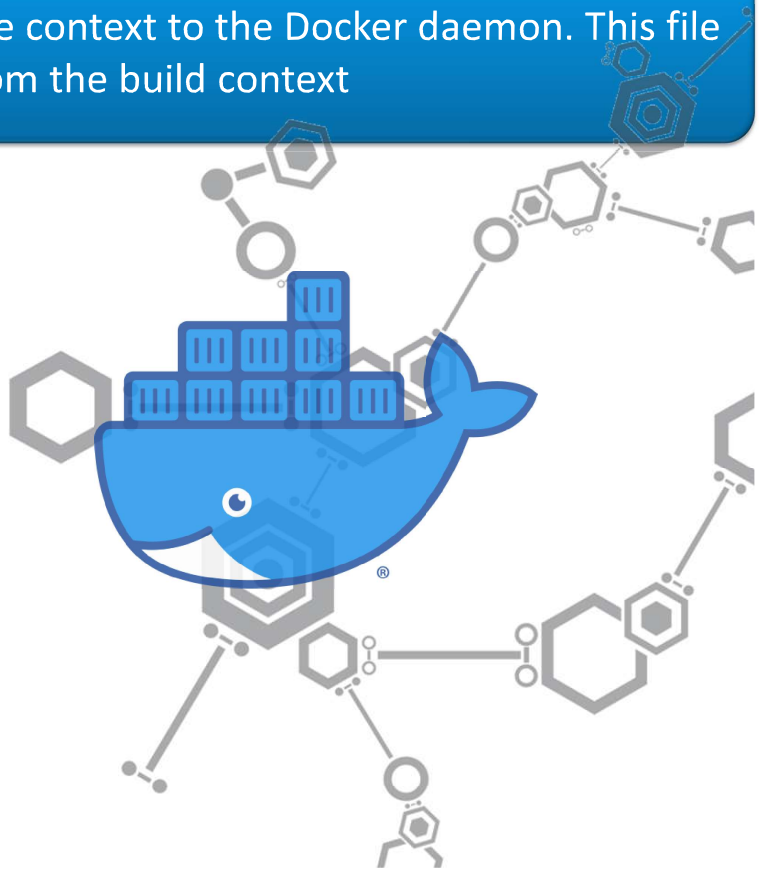


edureka!

Dockerfile: .dockerignore File

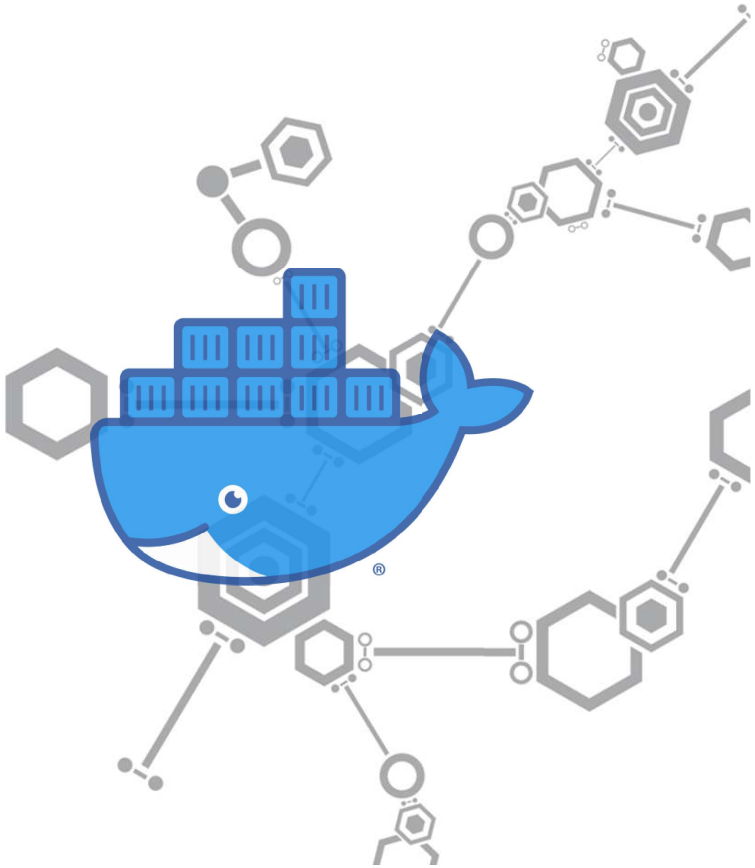
The Docker CLI refers to a file named .dockerignore before it sends the context to the Docker daemon. This file defines all the files and directories to exclude from the build context

- This also stops Docker from adding these files using the ADD/COPY instruction
- Comments can be added to the .dockerignore file using the `'#'` symbol
- The `'!'` mark can be used to make exceptions to the exclusions



Dockerfile: The Build Command

- The build command is handled by the daemon and not the CLI
- To exclude files while executing COPY instruction, use .dockerignore file in the context directory
- Use the `-t` flag to specify the repository and tag the image
- The `-t` flag can also be used to store the image in multiple repositories



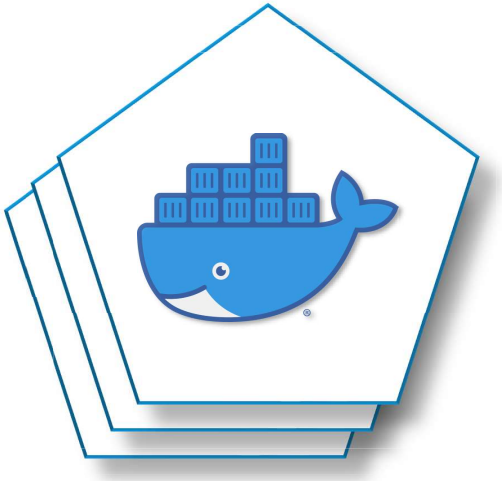
Dockerfile: CMD vs Entrypoint

CMD	Entrypoint
CMD defines the default command to execute inside the container	Entrypoint allows user to define how the container will run as an executable
CMD commands can be easily overridden at runtime	Entrypoint commands cannot be overridden unless the --entrypoint flag is added
If multiple CMDs are given inside a Dockerfile, only the last one is executed	If a CMD is added after Entrypoint, both execute in order
Example in shell format: `CMD echo "Docker CMD"`	Example in exec format: `ENTRYPOINT ["echo", "DCA"]`

edureka!

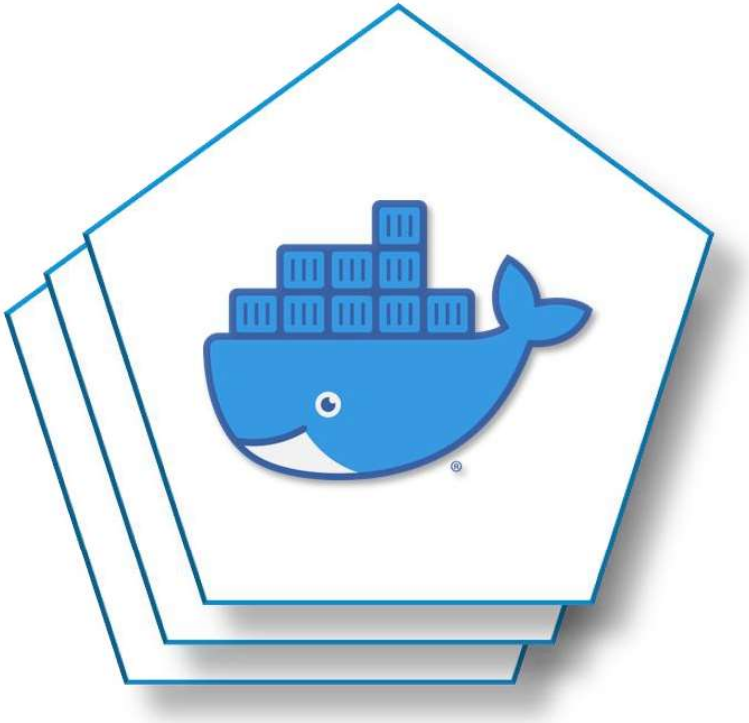
Docker Image

- Docker image is a file, consisting of multiple layers, that is executed to create a container
- The image is built from the instructions in the Dockerfile
- Containers use images to construct a run-time environment



Docker Base Image

- Docker base image is the basic image on top of which layers are added
- Docker tracks changes by adding a new image layer over the base image by using the Union File System (UFS)
- Example: To run the LAMP Stack, the user will require a base image of Linux OS and then subsequent layers of Apache, MySQL, and PHP are added on top



Building a Docker Image: Creating Dockerfile

```
root@test01:~# touch Dockerfile
root@test01:~# gedit Dockerfile
```

Dockerfile

```
1  FROM ubuntu
2  CMD ["echo", "hello world"]
```

FROM specifies the base image. In this scenario, it is the ubuntu image on the Docker hub repository

The layer above the base image. Here it executes the given command

Building a Docker Image: Building the Dockerfile

```
root@test01:~# gedit Dockerfile
root@test01:~# docker build .
Sending build context to Docker daemon   513 kB
Step 1 : FROM ubuntu
 ---> 4ca3a192ff2a
Step 2 : CMD echo hello world
 ---> Running in 876177664b4f
 ---> 7c226dc91bb2
Removing intermediate container 876177664b4f
Successfully built 7c226dc91bb2
root@test01:~#
```

Running hello-world in intermediate container

Final image ID

```
root@test01:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              7c226dc91bb2        8 minutes ago       128.2 MB
hello-world         latest              48b5124b2768        3 months ago        1.84 kB
ubuntu              latest              4ca3a192ff2a        4 months ago        128.2 MB
root@test01:~#
```

edureka!

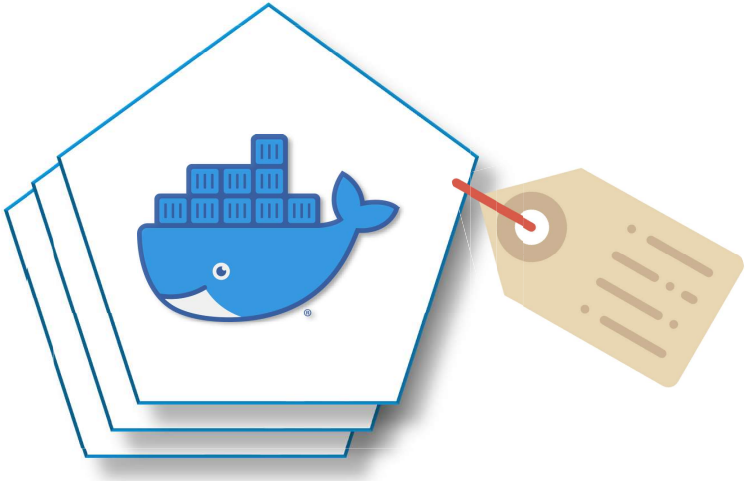
Building a Docker Image: Running the File

```
root@test01:~# docker run --name test 7c226dc91bb2
hello world
root@test01:~#
```

edureka!

Docker Image Tags

- Docker enables users to tag the image IDs with aliases which are easier to remember
- Tagging also enables users to convey useful information about the image through the name itself
- For example: tags can come in handy while adding a version number to the image



edureka!

Tags: Tagging an Image

Docker images can be tagged in one of two ways:

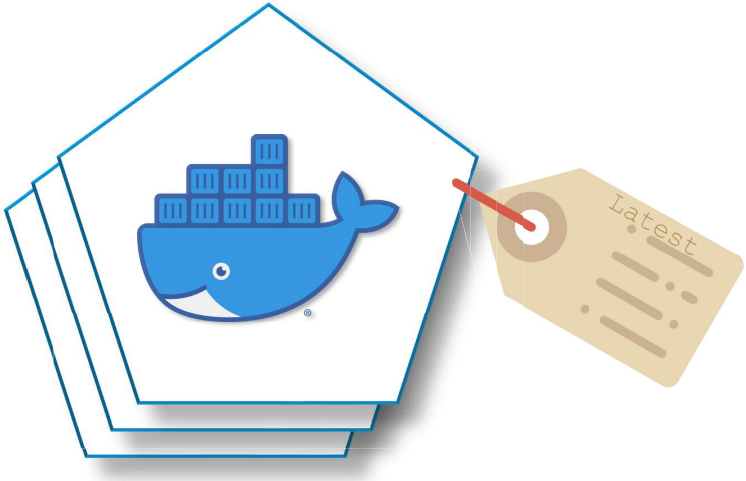
- While building the image from the Dockerfile
- Explicitly tagging the image using the tag command

```
# While Building an image
docker build . -t username/imageName:tagName
```

```
# Tagging an existing image
docker tag imageID username/imageName:tagName
```

Tags: Latest Tag

- If an image is not explicitly tagged then docker automatically provides it with the latest tag
- Example: if you create a myapp image, it can also be referred to as myapp:latest



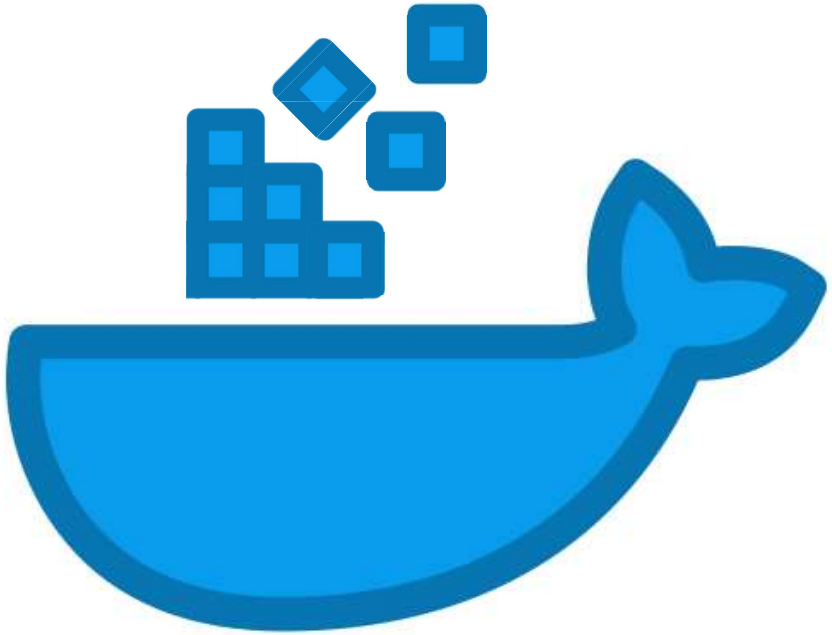
Managing Docker Images

- Docker images are portable and can be distributed amongst the entire organization making it very accessible
- The easiest way to make these images available to others is by using a Docker registry

edureka!

Docker Registry

- Docker Registry is a part of the Docker ecosystem
- A registry holds named Docker images for content delivery and storage
- The registry can be configured by creating a new configuration in YAML (Yet Another Markup Language) format



Managing Docker Images

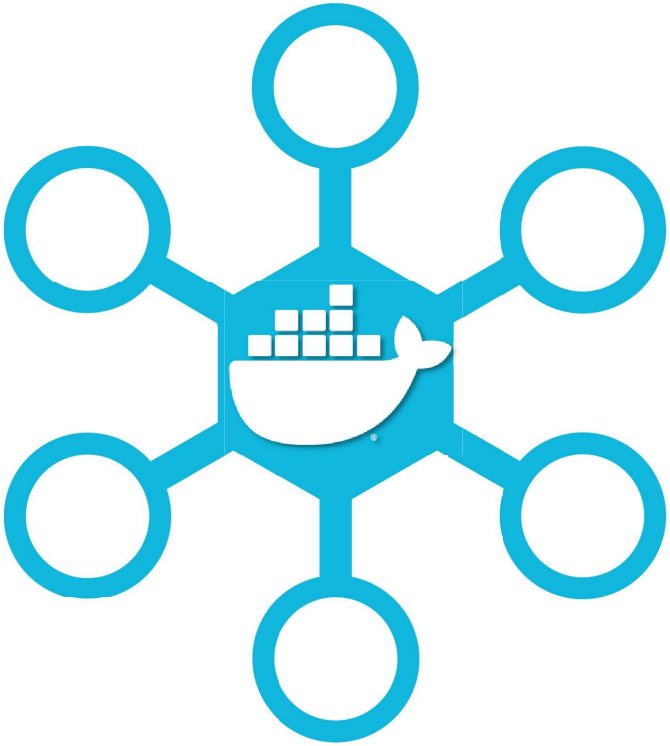
The images can be distributed using either of the following:

- Docker Hub (configured as default registry when docker is installed)
- Private registry
  - Quay.io
  - Google Container Registry
  - Artifactory
  - Amazon ECR Registry
  - Sonatype Nexus

Note: Docker Trusted Registry is taught in a separate module.

Docker Hub

- Docker Hub is a cloud-based docker registry
- It is a public repository for hosting, building and testing docker images
- It also provides a paid version which lets the user host private and team registries

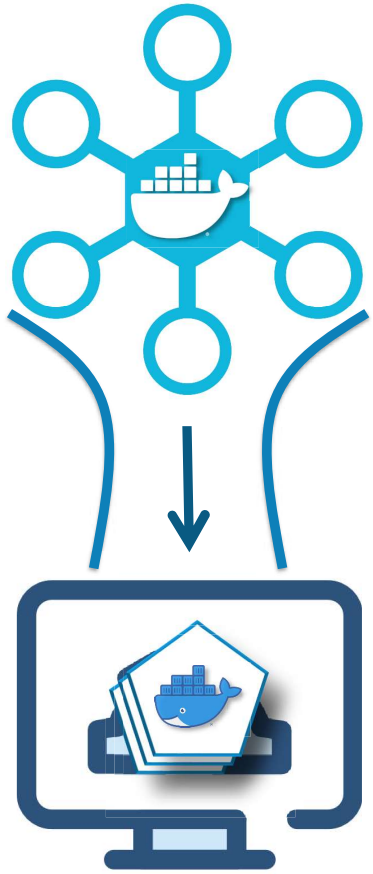


Pulling Images from Docker Hub

When the `docker run` command is executed:

- Docker searches for the corresponding image on the local system
- If not found, Docker automatically pulls the image from the Docker Hub registry to create the container
- Pulling an image from a private repository requires authentication

```
# In case, the user wants to explicitly pull an image but
# not run it, the docker pull command can be used
            docker pull registryName/imageName
```

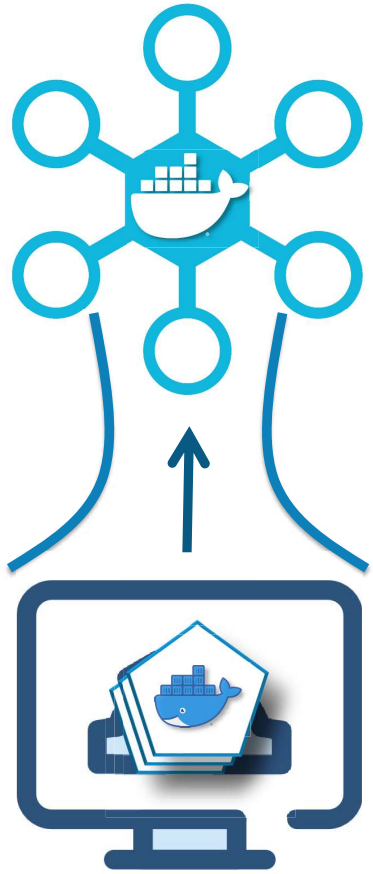


Pushing Images to Docker Hub

In order to push images to Docker Hub:

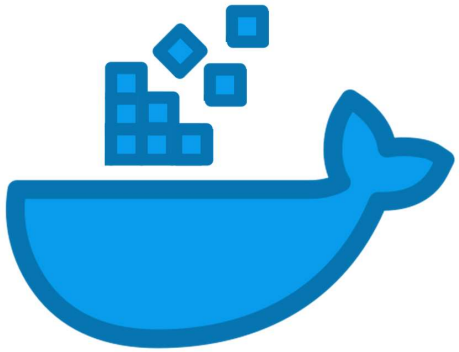
- Create a Docker Hub account
- Create a new repository
- Login to the repo from Docker CLI and push the image

```
# Log into Docker Hub from CLI using:
  docker login --username=yourUsername --email=yourEmail
# This will prompt a password response. Enter it and your
# credentials will be saved
# To push a custom image to Docker Hub use:
                docker push username/imageName
```



Deploying a Local Docker Registry

To start a registry on the local system, it first requires a registry container running locally



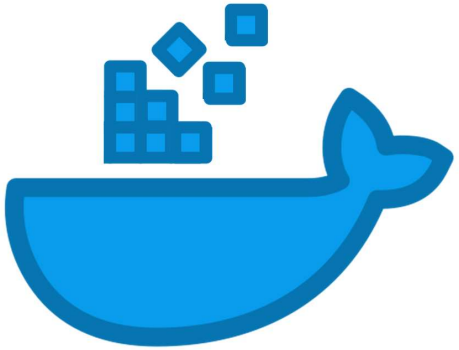
```
# The following command can be used to start a local registry container on port
# number 5000
        docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

edureka!

Using a Local Docker Registry

After deploying the local registry, users can push and pull images from it directly:

```
EXAMPLE:
# Tagging an existing image. Consider the image name is myapp
            docker tag myapp localhost:5000/my-app
# Pushing the image to the local registry
              docker push localhost:5000/my-app
# Pulling the image from the local repository
              docker pull localhost:5000/my-app
# To stop the local registry and delete its container
  docker container stop registry && container rm -v registry
```



Basic Configuration for Local Registry

Users can set basic configurations for the local registry while deploying it:

Configuration	Flag	Value
Starting a registry automatically with Docker	`--restart`	`always`
Switch the published port	`-p`	`<any available port on the system>`
Switch the registry port	`-e`	`REGISTRY_HTTP_ADDR=0.0.0.0:<newPort>`
Change the storage location using bind mount	`-v`	`/mnt/registry:/new/storage/directory`

edureka!

Running an External Registry

- Docker allows users to make their registries be available to the external audience
- This can be done by first securing the registry using TLS certificates
- Then, the Docker client must be configured to accept the new domain key and certificate