

# FULL STACK



## Docker Certified Associate Training

Source: <https://docs.docker.com>

# FULL STACK

## Image Creation, Management, and Registry





## Learning Objectives

By the end of this lesson, you will be able to:

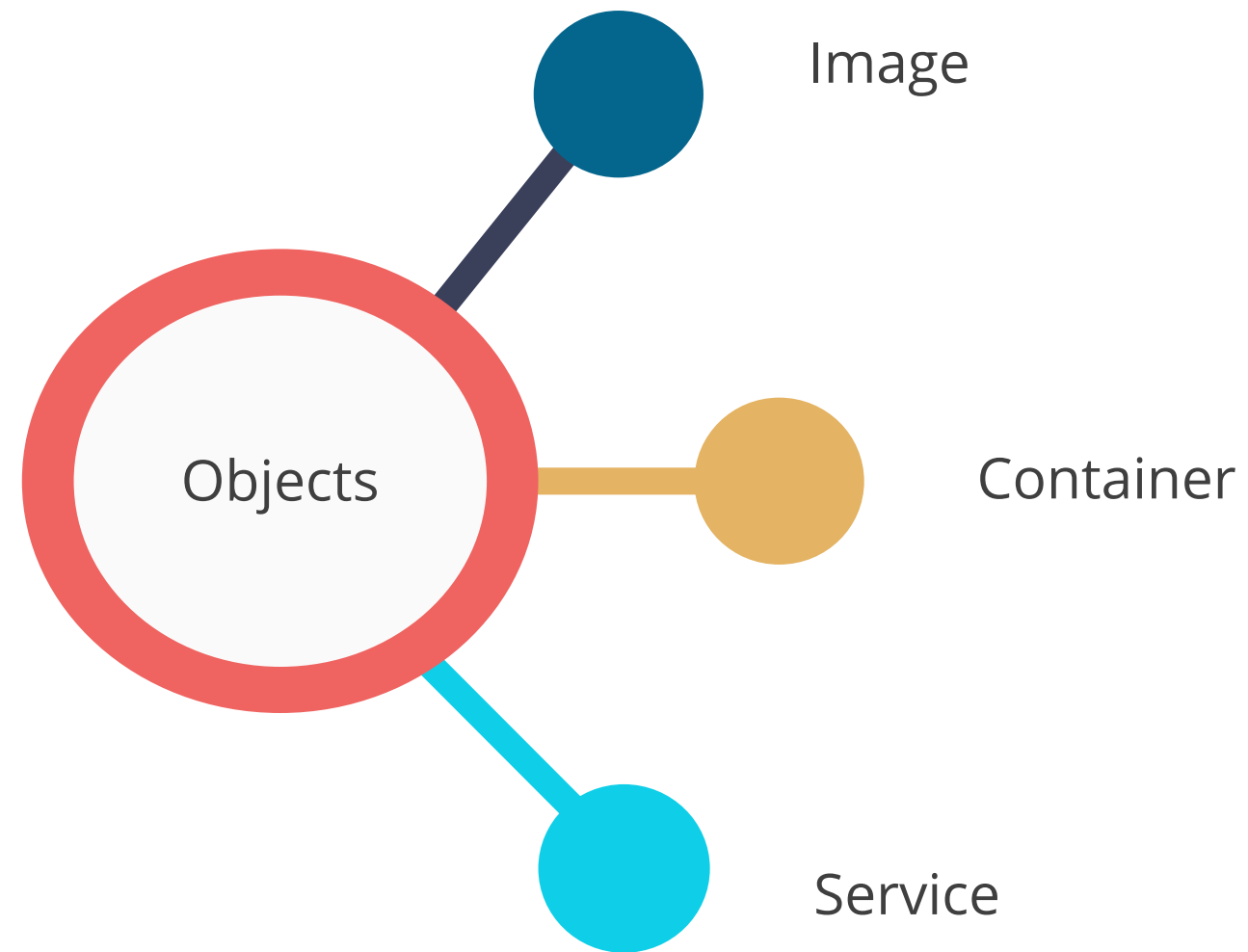
- Describe various Docker objects, such as image, container, and services
- Comprehend Dockerfile, its purpose in building images, and function of Dockerfile instructions
- Identify and describe the layers of an image that result from the instruction on a Dockerfile
- Comprehend the functionalities and purpose of the registries



# FULL STACK

## Objects

# Object Types

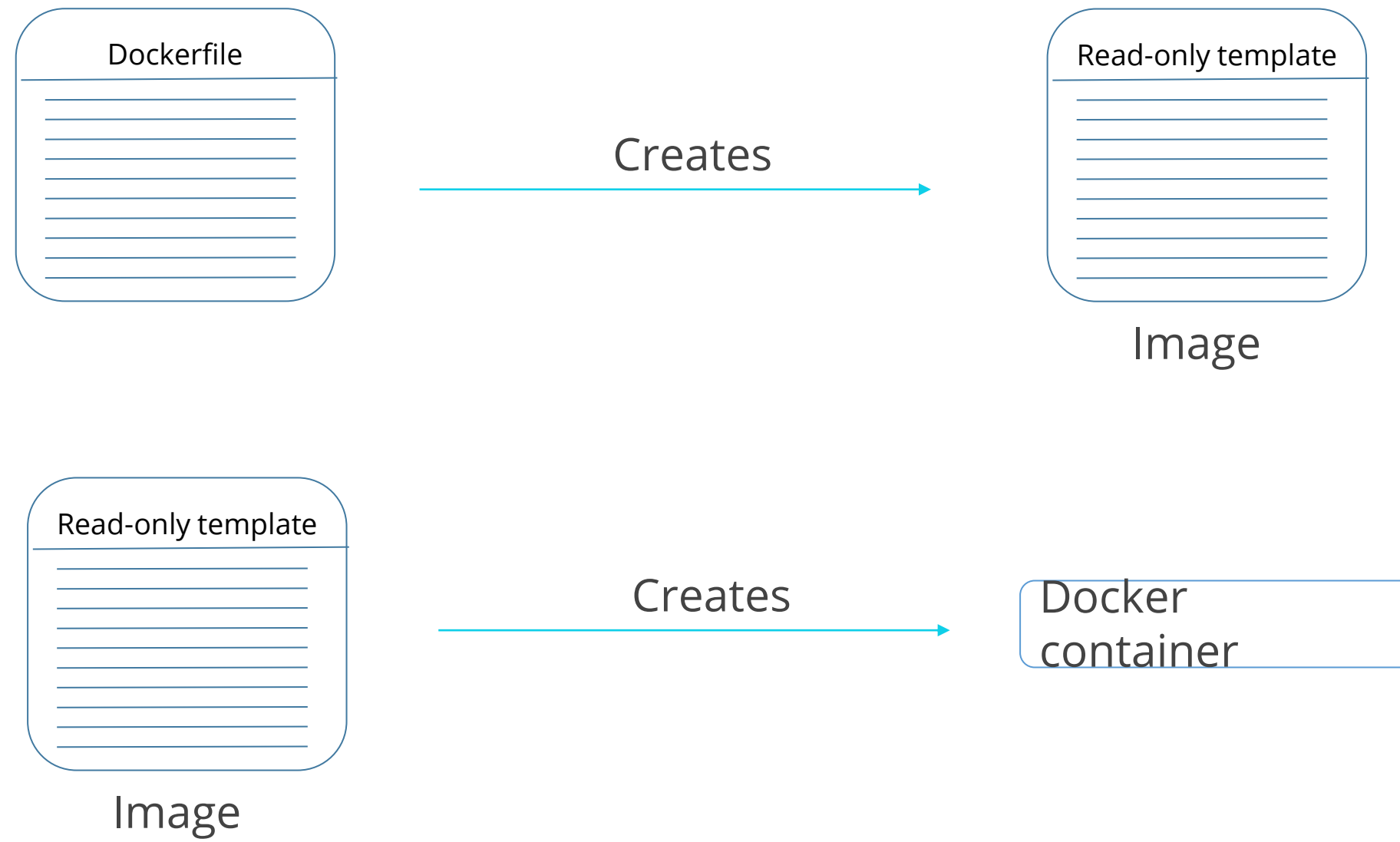


# FULL STACK

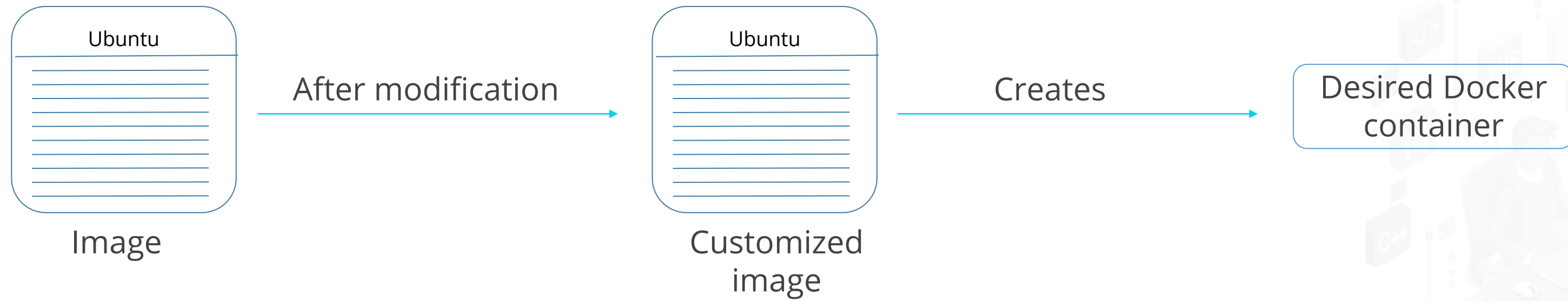
**Object: Image**

# Image: Overview

An image holds instructions that are required to run an application.

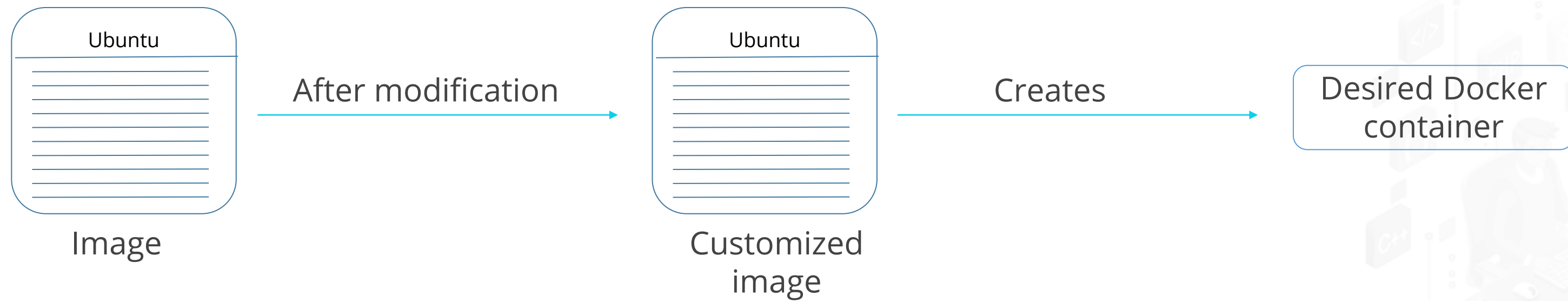


# Image: Overview

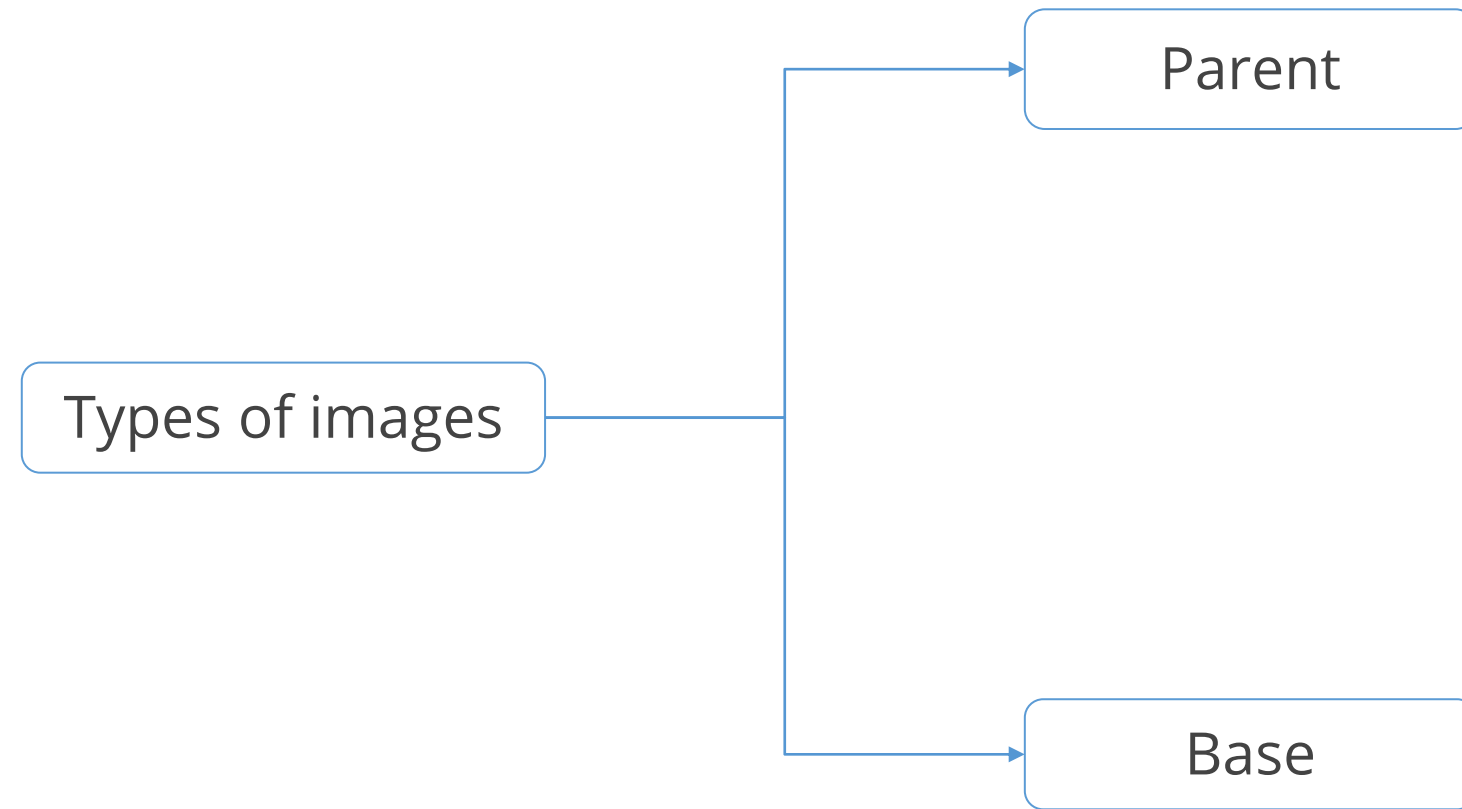




# Image: Overview



# Image: Overview



# FULL STACK

## Object: Container

# Container: Overview

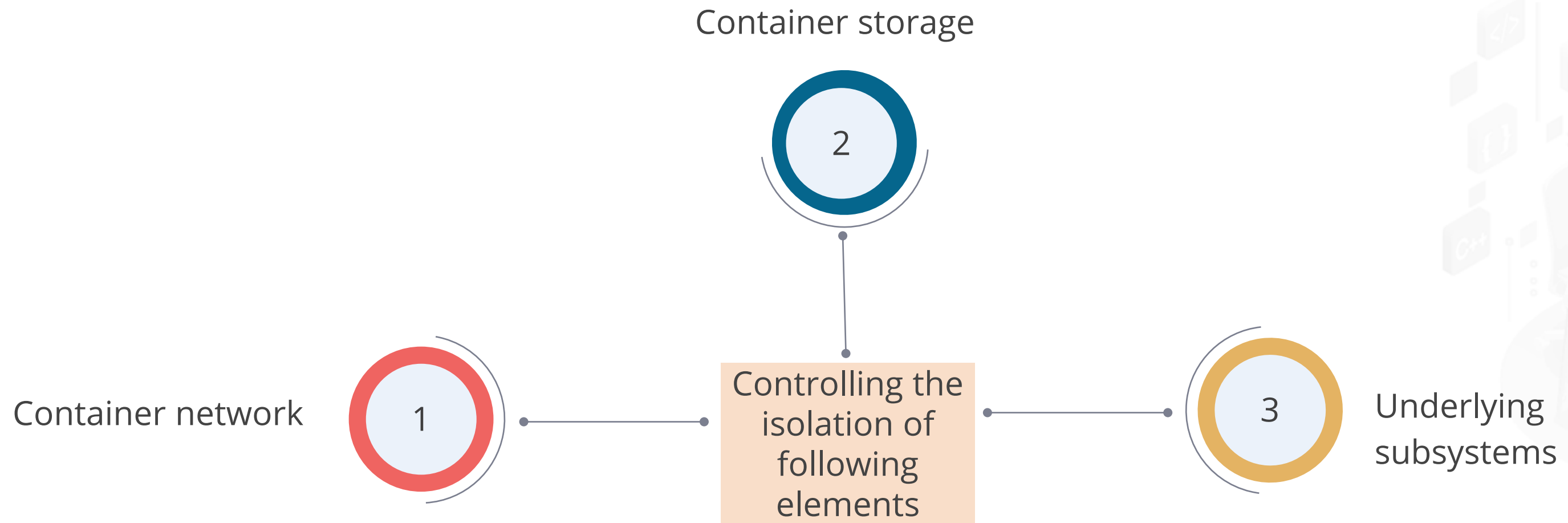
Runnable instance of an image is called as container.



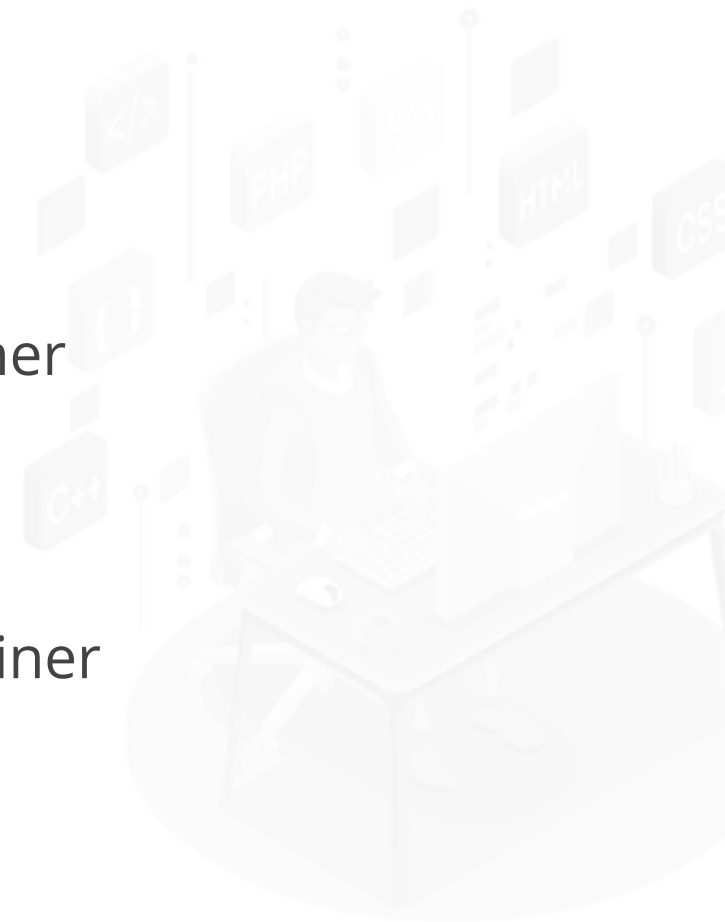
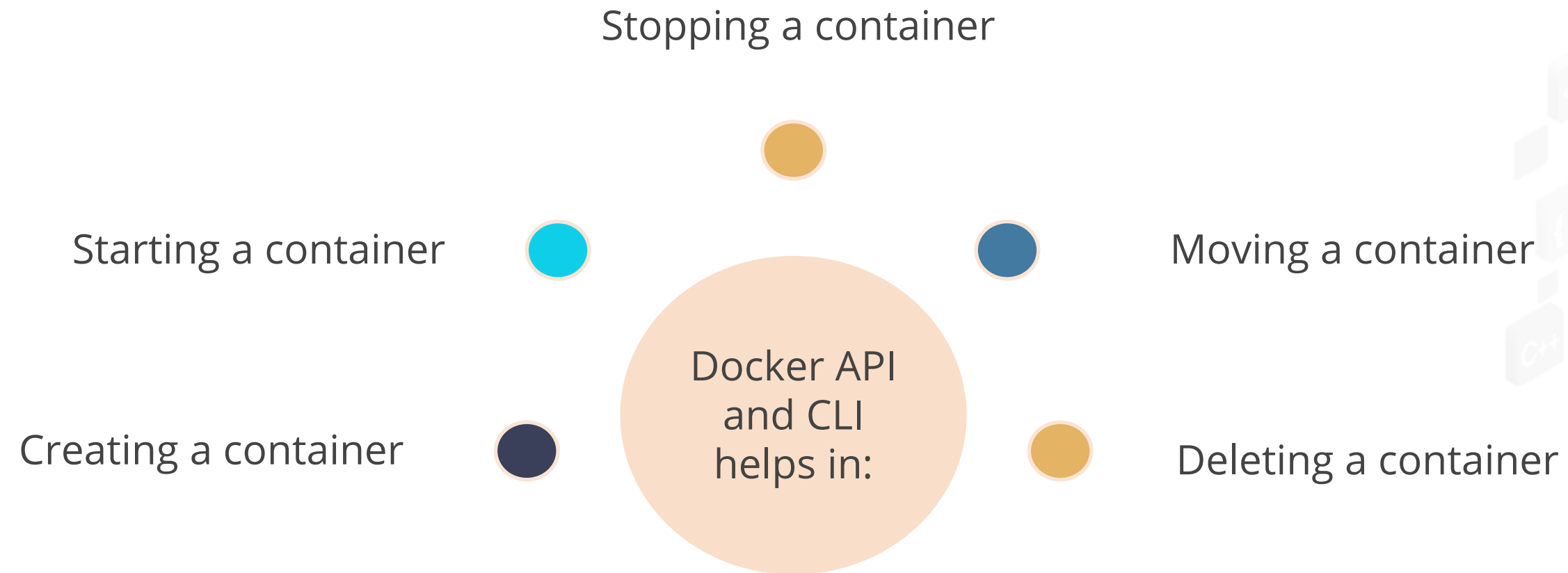


# Container: Overview

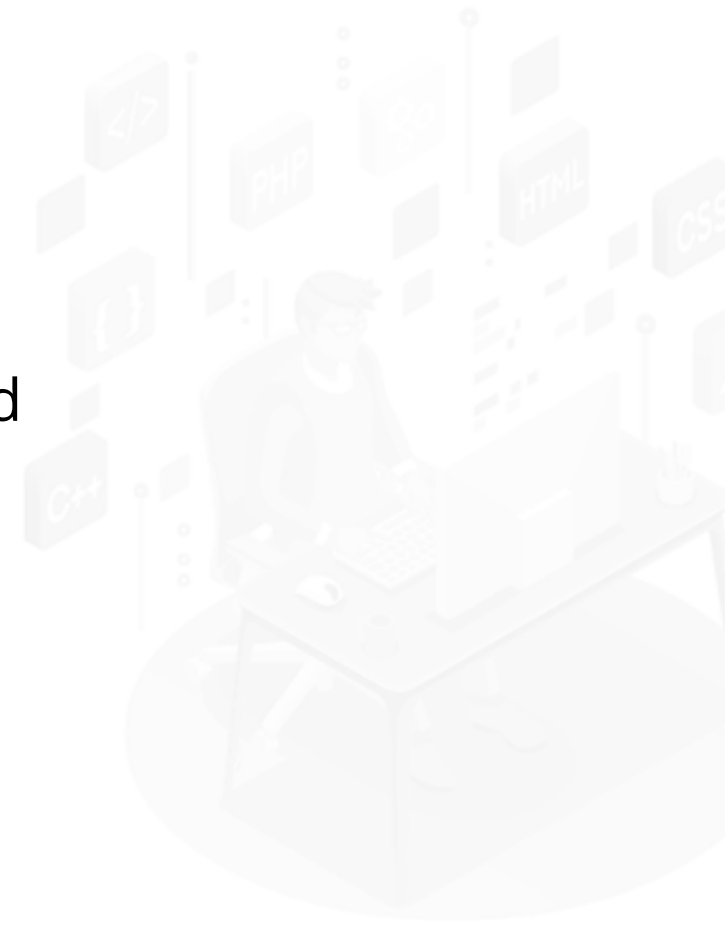
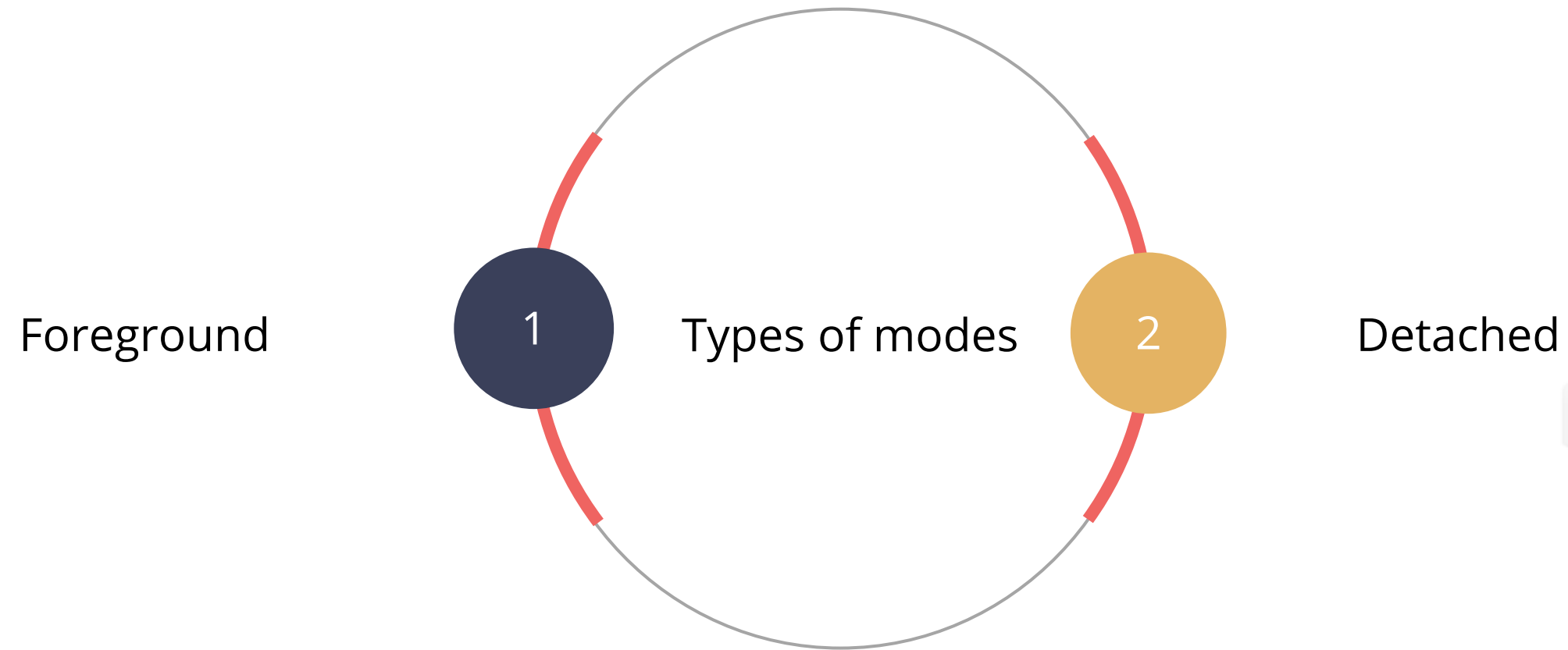
Inherently, containers are well isolated from one another and their host machines.



# Container: Overview

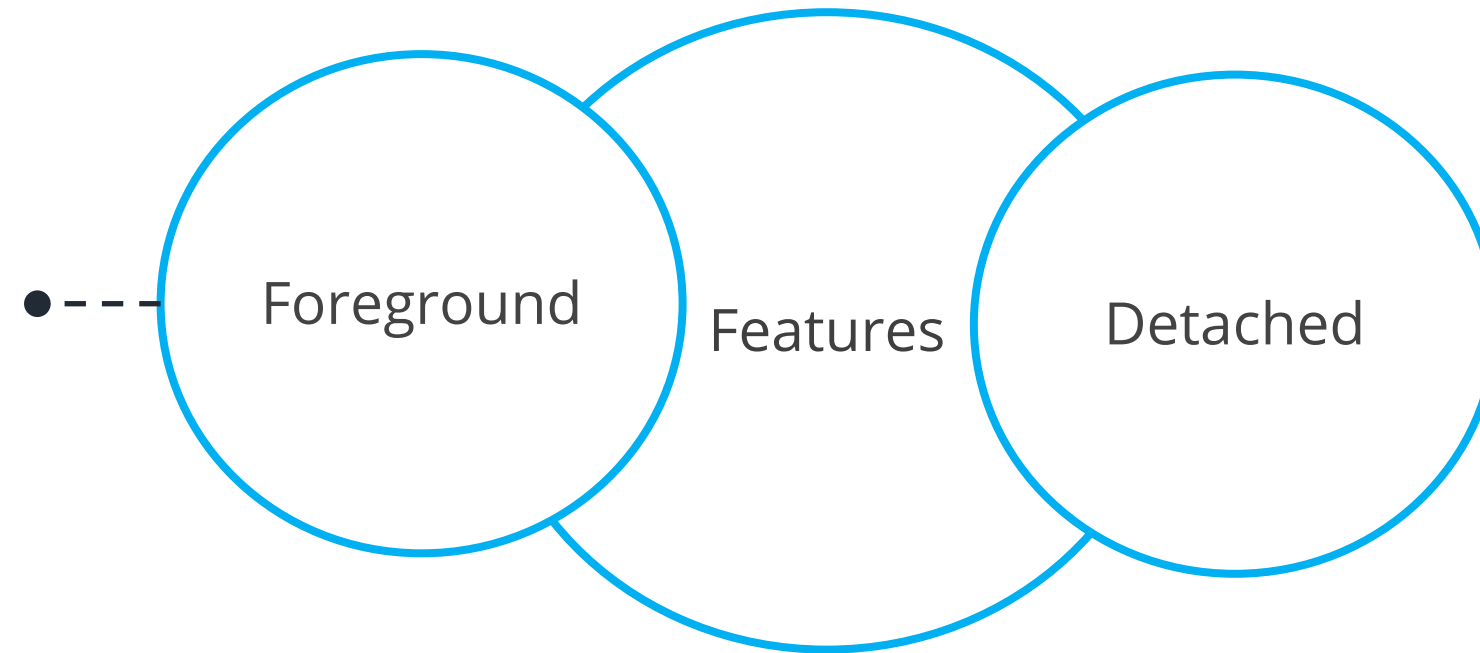


# Container Modes: Types



# Container Modes: Features

All the outputs are visible on the terminal. But, the container exits when the terminal is accessed or created again.



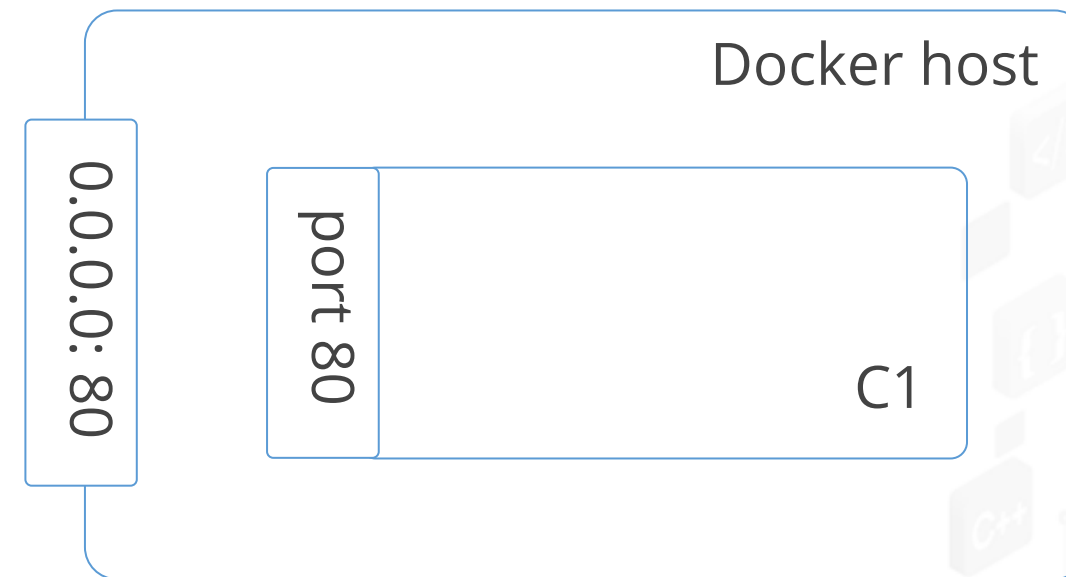
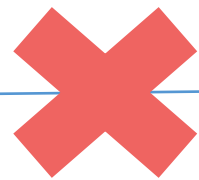
The output is not visible on the terminal. The container runs in the background when the terminal is accessed or created again.



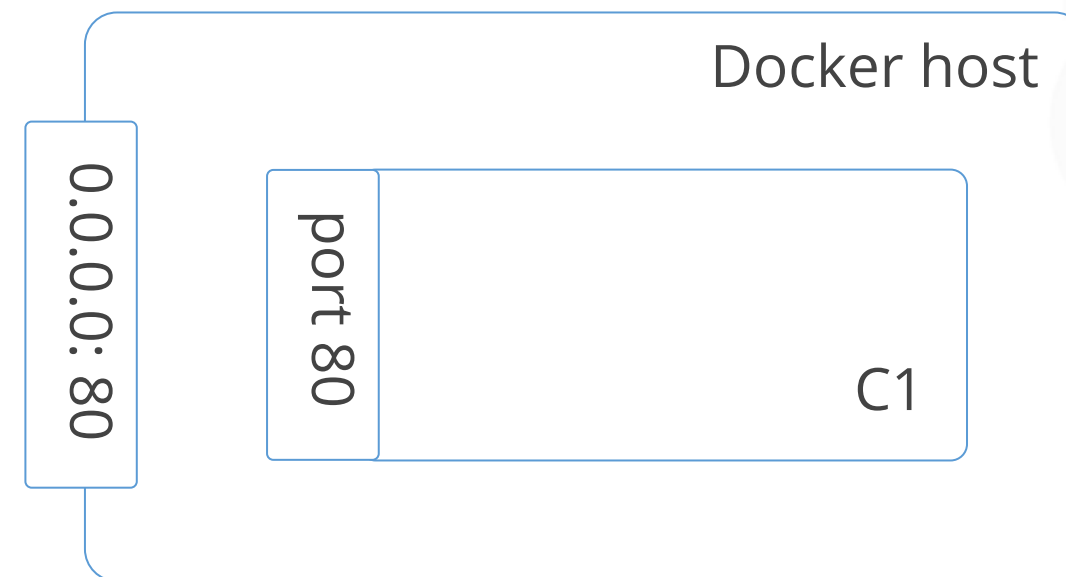
# Interaction

Interacting with container "C1":

Incoming connection at port 80



Incoming connection at 0.0.0.0:80 -> 80

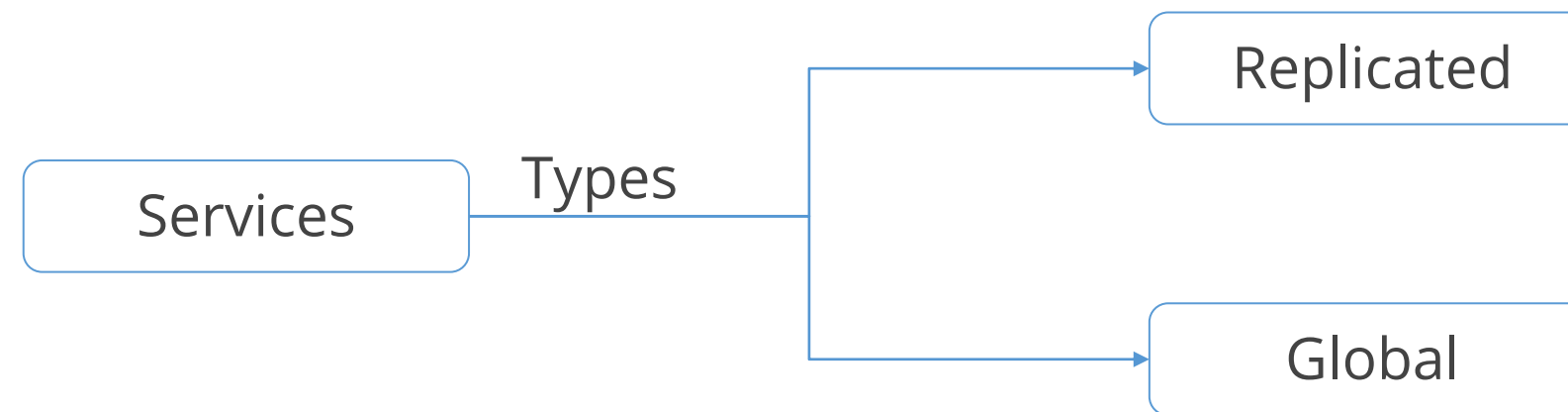


# FULL STACK

## Object: Service

# Services and Tasks: Overview

Services allow you to scale containers across multiple Docker daemons, which all work together as a swarm with multiple managers and workers. Each member of a swarm is a Docker daemon, and the daemons all communicate using the Docker API.



A service allows you to define the desired state, such as the number of replicas of the service that must be available at any given time. By default, the service is load-balanced across all worker nodes.

# Scheduling

Docker creates services that describe the desired state where work is done by tasks.

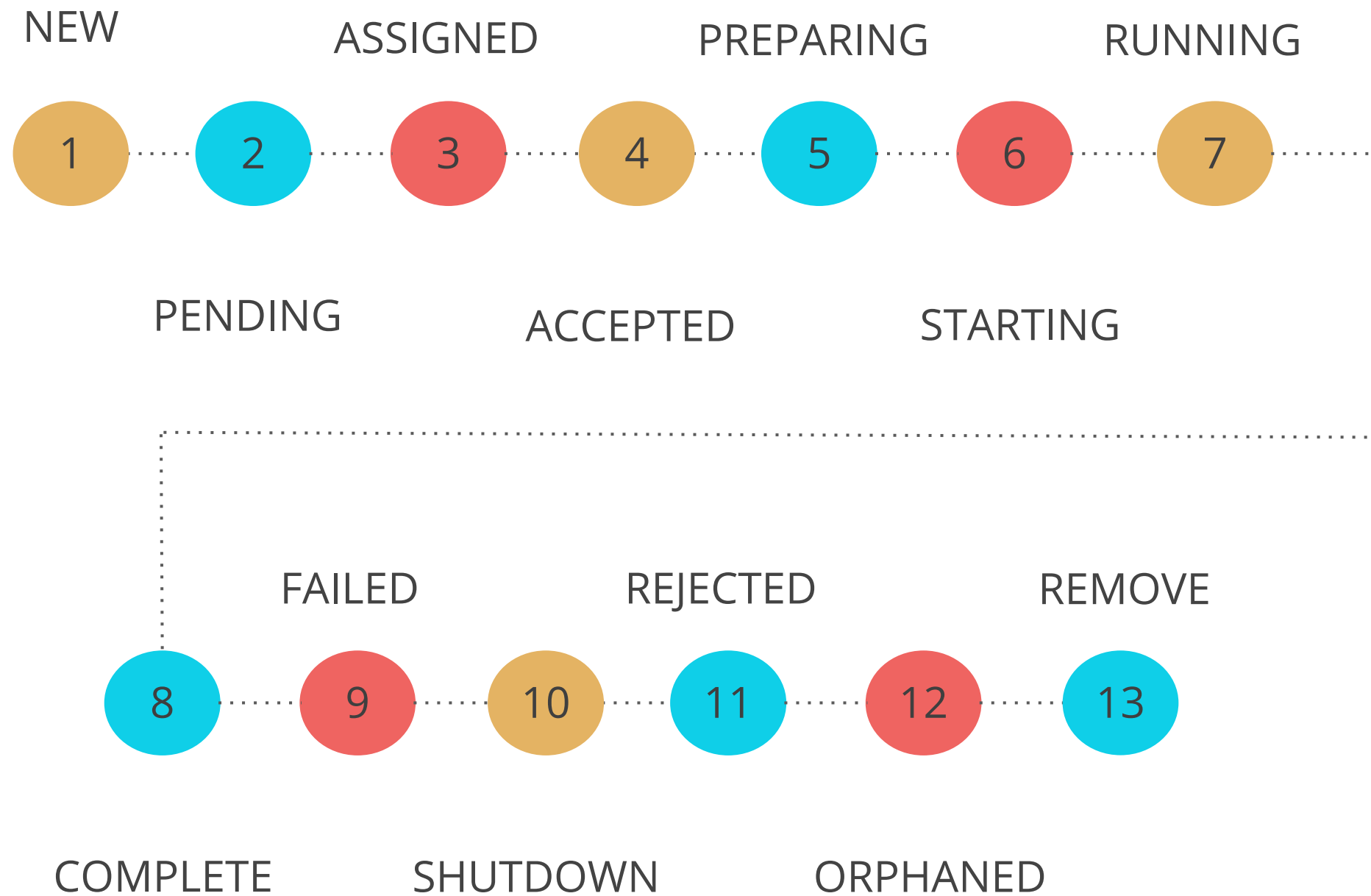
The sequence in which work is scheduled on swarm:

- 1 Service is created
- 2 Docker manager node receives the request
- 3 Docker manager nodes schedule the service
- 4 Services start the tasks
- 5 Tasks undergo various stages in its life cycle





# States of Tasks



# Commands

Create a container

*docker container create [OPTIONS] IMAGE [COMMAND] [ARG...]*

Create a container

*docker run -dt -p 80:80 nginx*

This command returns a UUID that helps in the identification of a container.

Naming a container  
using option *--name* in  
run command

*docker run -- name -dt -p 80:80 nginx*

Remove a container

*docker container rm [OPTIONS] CONTAINER [CONTAINER...]*

# Commands

Create a service

*docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]*

Remove a service

*docker service rm SERVICE [SERVICE...]*

View task state

*docker service ps <service-name>*

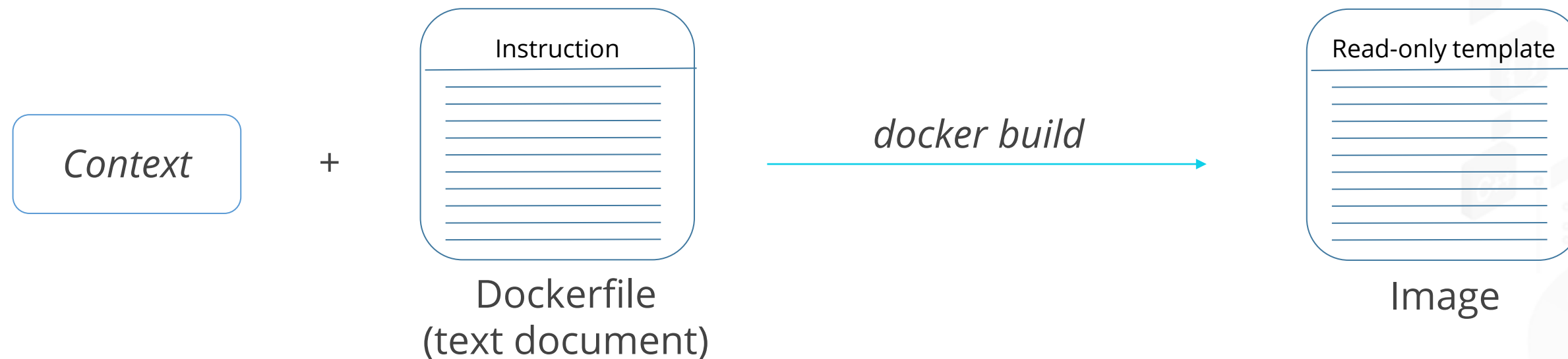
# FULL STACK

## Dockerfile and BuildKit



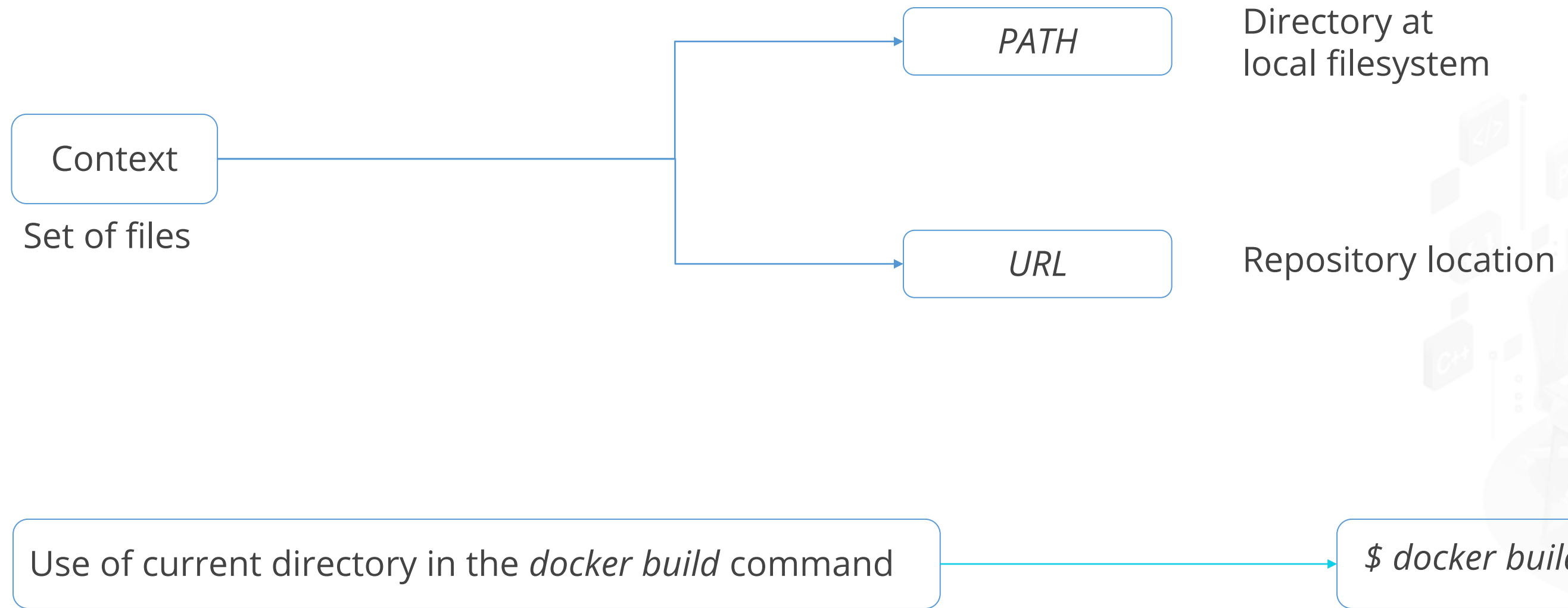
# Dockerfile: Overview

Dockerfile contains all the necessary instructions that are then used to build images.



*docker build* creates an image from a *context* and a *Dockerfile*.

# Dockerfile: Overview



# Format

#

Used for commenting and for  
parser/syntax/escape directives

INSTRUCTIONS

Conventionally they are written in uppercase

arguments

Not case-sensitive and **#** is treated as an  
argument

Example:

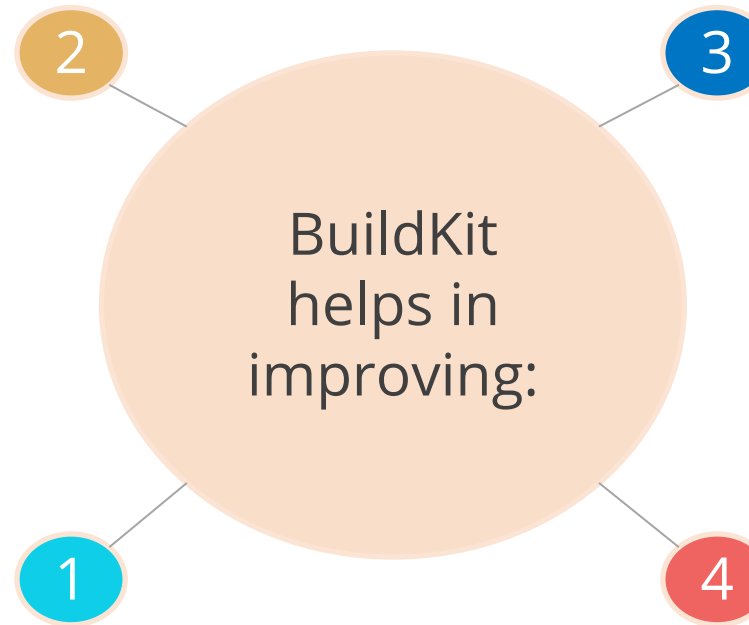
*# Comment*

*RUN echo 'welcome to the # barn of minas tirth'*

# BuildKit

BuildKit helps in converting the source code to build artifacts.

Storage management



Feature functionality

Performance

Security



# BuildKit

## Benefits provided by BuildKit:

Exempting the execution of unused build stages

1

Transferring the changed files in the build context between builds

2

Parallelizing the construction of independent build stages

3

Using the implementations of external Dockerfile along with various new features

4

Exempting the transfer of unused files in the build context

5

Eluding the side-effects with the help of API (intermediate images and containers)

6

Prioritizing the build cache for automatic pruning

7

# BuildKit

Setting the DOCKER\_BUILDKIT=1 environment variable

```
$ DOCKER_BUILDKIT=1 docker build .
```

Setting the daemon configuration in */etc/docker/daemon.json* feature to true

```
{ "features": { "buildkit": true } }
```

Location of the Dockerfile builder is defined by syntax directive

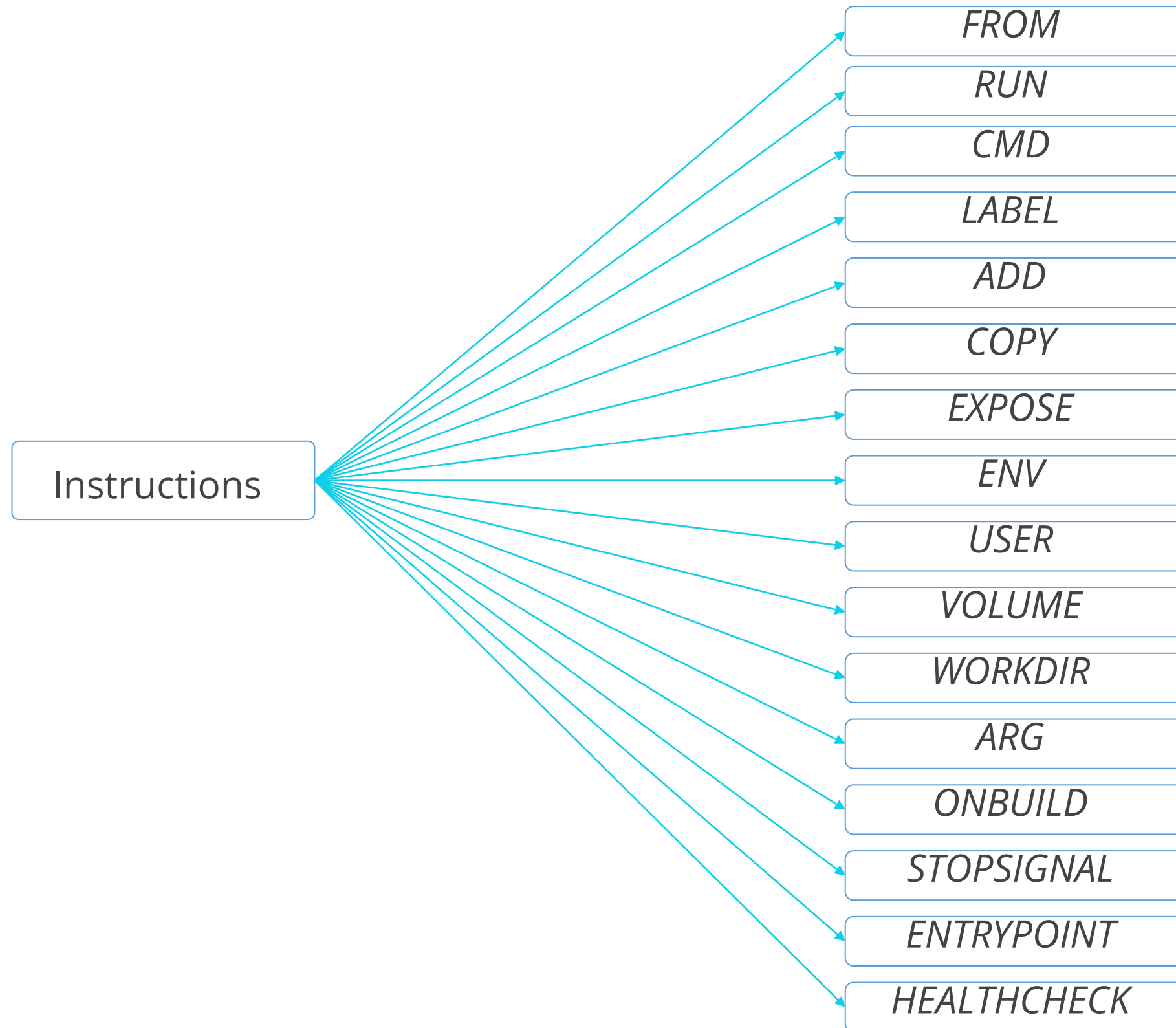
```
# syntax=[remote image reference]
```

# FULL STACK

## Instructions



# Instructions: List



# Instructions: List

*FROM*

*RUN*

*CMD*

*LABEL*

*ADD*

*COPY*

*EXPOSE*

*ENV*

*USER*

*VOLUME*

*WORKDIR*

*ARG*

*ONBUILD*

*STOPSIGNAL*

*ENTRYPOINT*

*HEALTHCHECK*

*FROM* is used to begin a new build stage. This sets the base image for subsequent instructions.

Properties:

- *FROM* can be written multiple times to create multiple images in a single Dockerfile.
- *name* is used to name the build stage in a Dockerfile.
- *FROM* is the first instruction in any Dockerfile. Only *ARG* can precede *FROM*, where *ARG* is outside the build stage.

Syntax:

*FROM* <image> [*AS* <name>]

# Instructions: List

FROM

**RUN**

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

*RUN* is used to execute commands inside the Docker image. These commands get executed once at build time and get written into the Docker image as a new layer.

Forms of *RUN* syntax:

shell form: *RUN* <command>

exec form: *RUN* ["executable", "param1", "param2"]

Note: “\” can be used in shell form to carry the *RUN* instruction to the next line.

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

*CMD* is used to define a default command to run when the container starts.

Forms of *CMD* syntax:

*exec form: CMD ["executable","param1","param2"]*

*default parameters: CMD ["param1","param2"]*

*shell form: CMD command param1 param2*

# Instructions: List

FROM

RUN

CMD

**LABEL**

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

*LABEL* is a key-value pair that helps in adding metadata to an image. One image can have multiple labels that can be written in a single instruction.

*LABEL* syntax:

*LABEL* <key>=<value> <key>=<value> <key>=<value>

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

*ADD* copies new directories, files, and URLs of remote files from *<src>*. It not only copies the directories, files, and URLs of remote files but also adds them to the image filesystem at *<dest>*.

The *<dest>* is a path where the source is copied in the destination container.

Forms of *ADD* syntax:

1. *ADD* [--chown=*<user>*:*<group>*] *<src>*... *<dest>*
2. *ADD* [--chown=*<user>*:*<group>*] ["*<src>*",... "*<dest>*"]

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

Rules obeyed by *ADD*:

1. The path of *<src>* requires to be included in the *context* of build.
2. The following occurs when the *<src>* is a URL and the *<dest>* does not end with a trailing slash:
  - The file is downloaded from the URL and is copied to the *<dest>*.
3. The following occurs when the *<src>* is a URL and the *<dest>* ends with a trailing slash:
  - The filename is inferred from the URL and the file gets downloaded to the *<dest>/<filename>*.
4. The contents of the directory and filesystem metadata get copied when the *<src>* acts as a directory.
5. The *<src>* is unpacked as a directory when *<src>* acts as *local* tar archive in the various compression formats. The compression format can be identity, gzip, bzip2, or xz.



# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

Rules obeyed by *ADD*:

6. The `<src>` file is copied along with its metadata if the `<src>` is present in any other format. The `<dest>` acts as a directory. This holds the contents of `<src>` that are written at `<dest>/base(<src>)`, which happens only when the `<dest>` ends with the trailing slash.
7. The `<dest>` must act as a directory that end with a slash `"/` when multiple `<src>` resources are specified.
8. The `<dest>` is considered as a regular file. The `<src>` content is written at `<dest>` when the `<dest>` does not end with a trailing slash.
9. The `<dest>` is created along with all the missing directories when the `<dest>` does not exist.

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

**COPY**

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

The *COPY* copies new directories and files from *<src>*. *COPY* also adds those new directories and files to the container's filesystem at the path *<dest>*.

Forms of *COPY*:

1. *COPY* [--chown=*<user>*:*<group>*] *<src>*... *<dest>*
2. *COPY* [--chown=*<user>*:*<group>*] ["*<src>*",... "*<dest>*"]

Note: The *--chown* feature is only supported on Dockerfiles that are used to build Linux containers. This feature is used to change the ownership of the copied directory.

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

Rules obeyed by *COPY*:

1. The path of `<src>` requires to be included in the *context* of build.
2. The contents of the directory and filesystem metadata get copied when the `<src>` acts as a directory.
3. The `<src>` file is copied along with its metadata if the `<src>` is present in any other format. The `<dest>` is a directory where the `<src>` contents are written at `<dest>/base(<src>)`, this happens only when the `<dest>` ends with a trailing slash.
4. The `<dest>` must act as a directory and end with a slash `"/"` when multiple `<src>` resources are specified.
5. The `<dest>` is considered as a regular file and the `<src>` content is written at `<dest>` when the `<dest>` does not end with a trailing slash.
6. The `<dest>` is created along with all the missing directories when the `<dest>` does not exist.

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

*EXPOSE* sends information to Docker that the container listens on the specified network ports at runtime.

*EXPOSE* syntax:

*EXPOSE* <port> [<port>/<protocol>...]

Note: *EXPOSE* does not publish any ports.

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

*ENV* is a key-value pair. It sets the *<key>*, an environment variable, to the value *<value>*.

*ENV* forms:

1. *ENV <key> <value>*
2. *ENV <key>=<value>*

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

**USER**

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

*USER* assigns user name and user group while running the image. It also assigns user name and user group for the RUN, CMD, and ENTRYPOINT instructions.

*USER* syntax:

1. *USER* <user>[:<group>] or
2. *USER* <UID>[:<GID>]

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

*VOLUME* creates a mount point with a specific name.

*VOLUME* syntax:

*VOLUME ["/data"]*



# Instructions: List

*FROM*

*RUN*

*CMD*

*LABEL*

*ADD*

*COPY*

*EXPOSE*

*ENV*

*USER*

*VOLUME*

*WORKDIR*

*ARG*

*ONBUILD*

*STOPSIGNAL*

*ENTRYPOINT*

*HEALTHCHECK*

*WORKDIR* sets the directory for RUN, CMD, ENTRYPOINT, COPY, and ADD instructions.

*WORKDIR* syntax:

*WORKDIR /path/to/workdir*

# Instructions: List

*FROM*

*RUN*

*CMD*

*LABEL*

*ADD*

*COPY*

*EXPOSE*

*ENV*

*USER*

*VOLUME*

*WORKDIR*

*ARG*

*ONBUILD*

*STOPSIGNAL*

*ENTRYPOINT*

*HEALTHCHECK*

*ARG* defines the variables that are passed by the user to the builder at the build-time.

*ARG* syntax:

*ARG* <name>[=<default value>]

# Instructions: List

*FROM*

*RUN*

*CMD*

*LABEL*

*ADD*

*COPY*

*EXPOSE*

*ENV*

*USER*

*VOLUME*

*WORKDIR*

*ARG*

*ONBUILD*

*STOPSIGNAL*

*ENTRYPOINT*

*HEALTHCHECK*

Default values:

*ARG* includes default value.

For example:

*FROM busybox*

*ARG user1=someuser*

*ARG buildno=1*



# Instructions: List

*FROM*

*RUN*

*CMD*

*LABEL*

*ADD*

*COPY*

*EXPOSE*

*ENV*

*USER*

*VOLUME*

*WORKDIR*

*ARG*

*ONBUILD*

*STOPSIGNAL*

*ENTRYPOINT*

*HEALTHCHECK*

Scope:

*ARG* defines the variables. This definition becomes effective from the line on which it is defined in the Dockerfile.

For example:

*FROM busybox*

*USER \${user:-some\_user}*

*ARG user*

*USER \$user*

# Instructions: List

*FROM*

*RUN*

*CMD*

*LABEL*

*ADD*

*COPY*

*EXPOSE*

*ENV*

*USER*

*VOLUME*

*WORKDIR*

*ARG*

*ONBUILD*

*STOPSIGNAL*

*ENTRYPOINT*

*HEALTHCHECK*

Variables:

*ARG* specifies variables available to the *RUN* instruction.

*ARG* syntax:

*FROM ubuntu*

*ARG CONT\_IMG\_VER*

*RUN echo \$CONT\_IMG\_VER*

# Instructions: List

*FROM*

*RUN*

*CMD*

*LABEL*

*ADD*

*COPY*

*EXPOSE*

*ENV*

*USER*

*VOLUME*

*WORKDIR*

*ARG*

*ONBUILD*

*STOPSIGNAL*

*ENTRYPOINT*

*HEALTHCHECK*

Predefined *ARG* variables:

There are variables that can be used without writing an *ARG* instruction in the Dockerfile.

Variable list:

- *HTTP\_PROXY*
- *http\_proxy*
- *HTTPS\_PROXY*
- *https\_proxy*
- *FTP\_PROXY*
- *ftp\_proxy*
- *NO\_PROXY*
- *no\_proxy*

# Instructions: List

*FROM*

*RUN*

*CMD*

*LABEL*

*ADD*

*COPY*

*EXPOSE*

*ENV*

*USER*

*VOLUME*

*WORKDIR*

*ARG*

*ONBUILD*

*STOPSIGNAL*

*ENTRYPOINT*

*HEALTHCHECK*

Automatic platform *ARGs* variables:

- *TARGETPLATFORM*
- *TARGETOS*
- *TARGETARCH*
- *TARGETVARIANT*
- *BUILDPLATFORM*
- *BUILDOS*
- *BUILDARCH*
- *BUILDVARIANT*

Note: Automatic platform *ARGs* are available while using Buildkit backend.



# Instructions: List

*FROM*

*RUN*

*CMD*

*LABEL*

*ADD*

*COPY*

*EXPOSE*

*ENV*

*USER*

*VOLUME*

*WORKDIR*

*ARG*

*ONBUILD*

*STOPSIGNAL*

*ENTRYPOINT*

*HEALTHCHECK*

Impact on build caching:

*ARG* variables do not sustain permanently in the build image. A “cache miss” occurs when a Dockerfile defines an *ARG* variable with a different value from the previous build.

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

**ONBUILD**

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

*ONBUILD* adds a *trigger* instruction to an image when the image is used as the base for another build.

The *ADD* and *RUN* instructions cannot be used when the image is a reusable application builder, because there is no access to the application source code.

The solution to counter this is to use *ONBUILD* instructions in advance that can be run during the next build stage.

*ONBUILD* syntax:

[...]

*ONBUILD ADD . /app/src*

*ONBUILD RUN /usr/local/bin/python-build --dir /app/src*

[...]

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

**STOPSIGNAL**

ENTRYPOINT

HEALTHCHECK

*STOPSIGNAL* sends a system call signal that helps the container to exit.

*STOPSIGNAL* syntax:

*STOPSIGNAL signal*

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

*ENTRYPOINT* helps in configuring an executable container.

Forms of *ENTRYPOINT*:

1. *exec* form: *ENTRYPOINT* ["executable", "param1", "param2"]
2. *shell* form: *ENTRYPOINT* command param1 param2

It will execute in */bin/sh -c*. This form ignores the *CMD* and *docker run* command line arguments.

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

Rules for CMD and ENTRYPOINT cooperation:

1. *CMD* or *ENTRYPOINT* commands must be specified in the Dockerfile.
2. *CMD* must be used to define default arguments for an *ENTRYPOINT* command.
3. *CMD* is overridden when the container is run with alternative arguments.
4. *ENTRYPOINT* must be defined while using an executable container.

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

*HEALTHCHECK* helps to identify whether applications are working in the desired fashion or not.

Forms of *HEALTHCHECK* :

1. *HEALTHCHECK [OPTIONS] CMD command*: This command helps in checking the container health by running a command inside it.
2. *HEALTHCHECK NONE*: This disables the health check that is inherited from the base image.

# Instructions: List

FROM

RUN

CMD

LABEL

ADD

COPY

EXPOSE

ENV

USER

VOLUME

WORKDIR

ARG

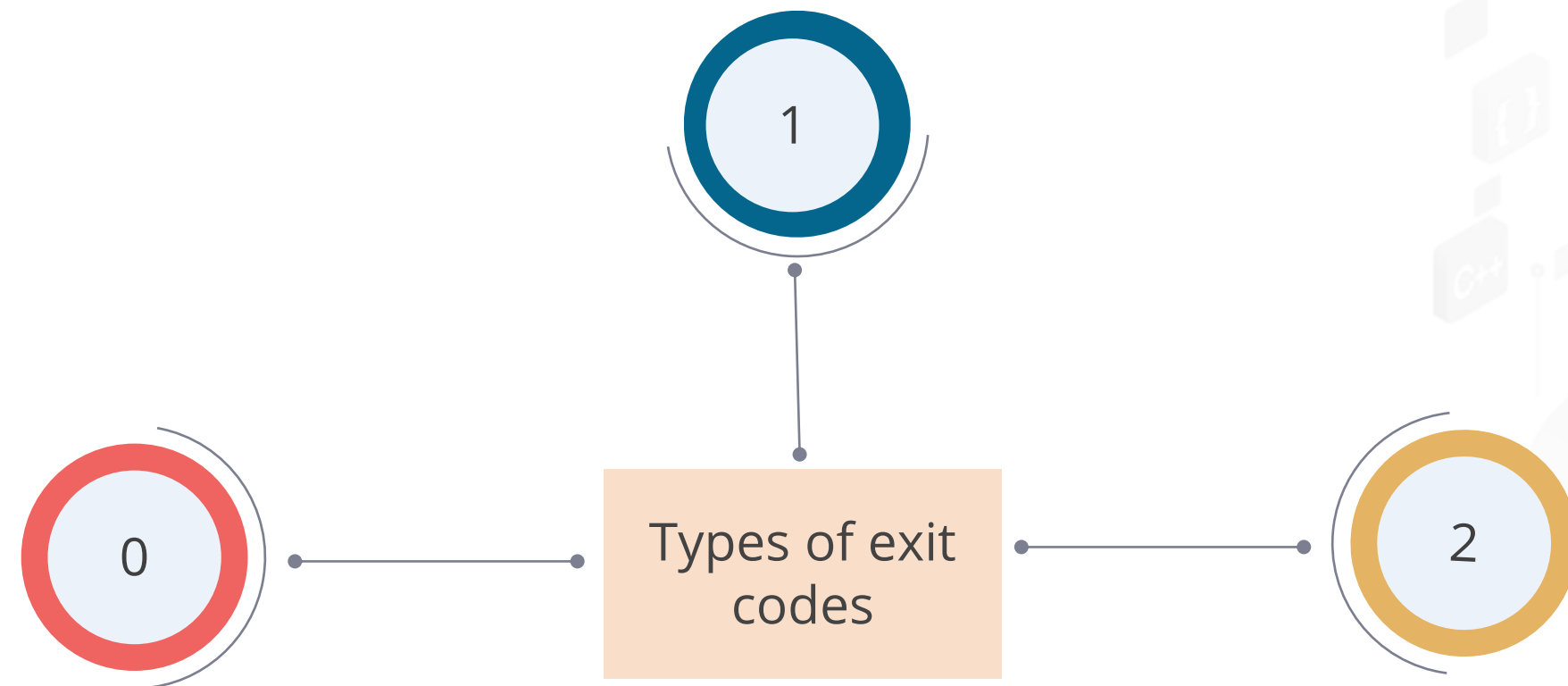
ONBUILD

STOPSIGNAL

ENTRYPOINT

HEALTHCHECK

*HEALTHCHECK* uses the value of exit codes to identify whether the applications are working properly or not.



# Create a Docker Image



**Problem Statement:** Your manager has asked you to create a Docker image from a dockerfile so that it can be shared with others.

## Steps to Perform:

1. Create a Dockerfile.
2. Use the dockerfile to create a Docker image.

ASSISTED PRACTICE

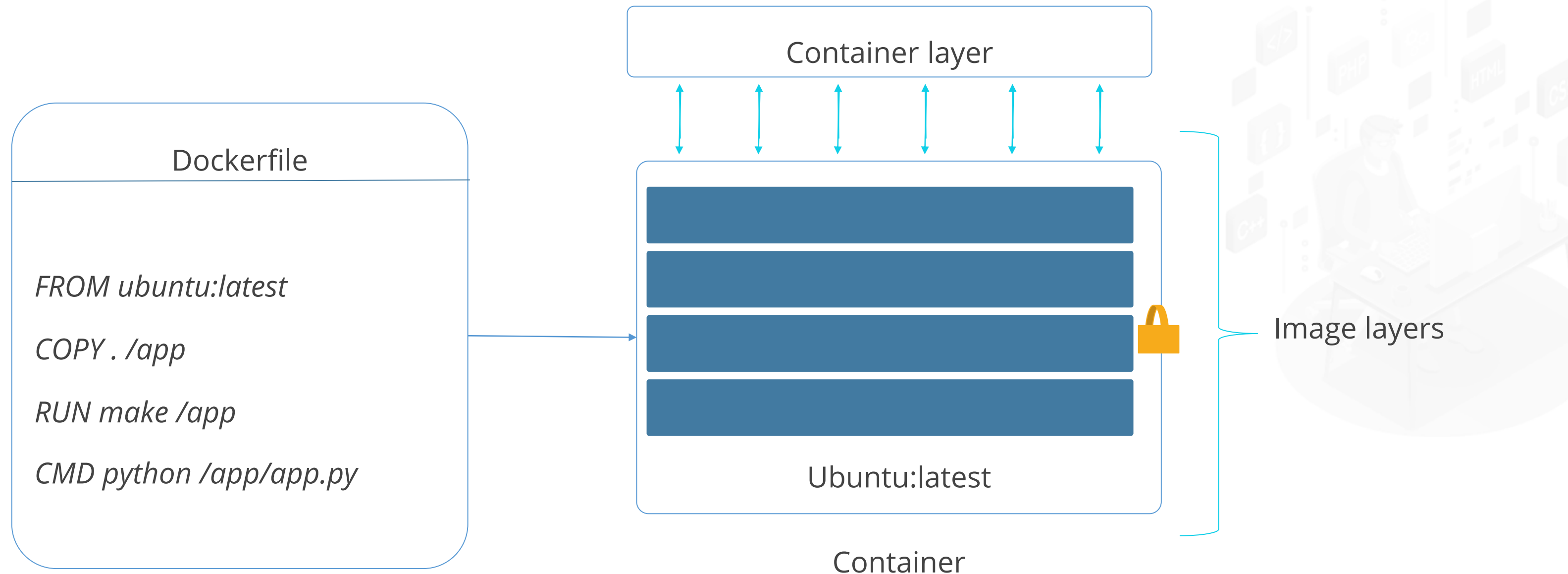


# FULL STACK

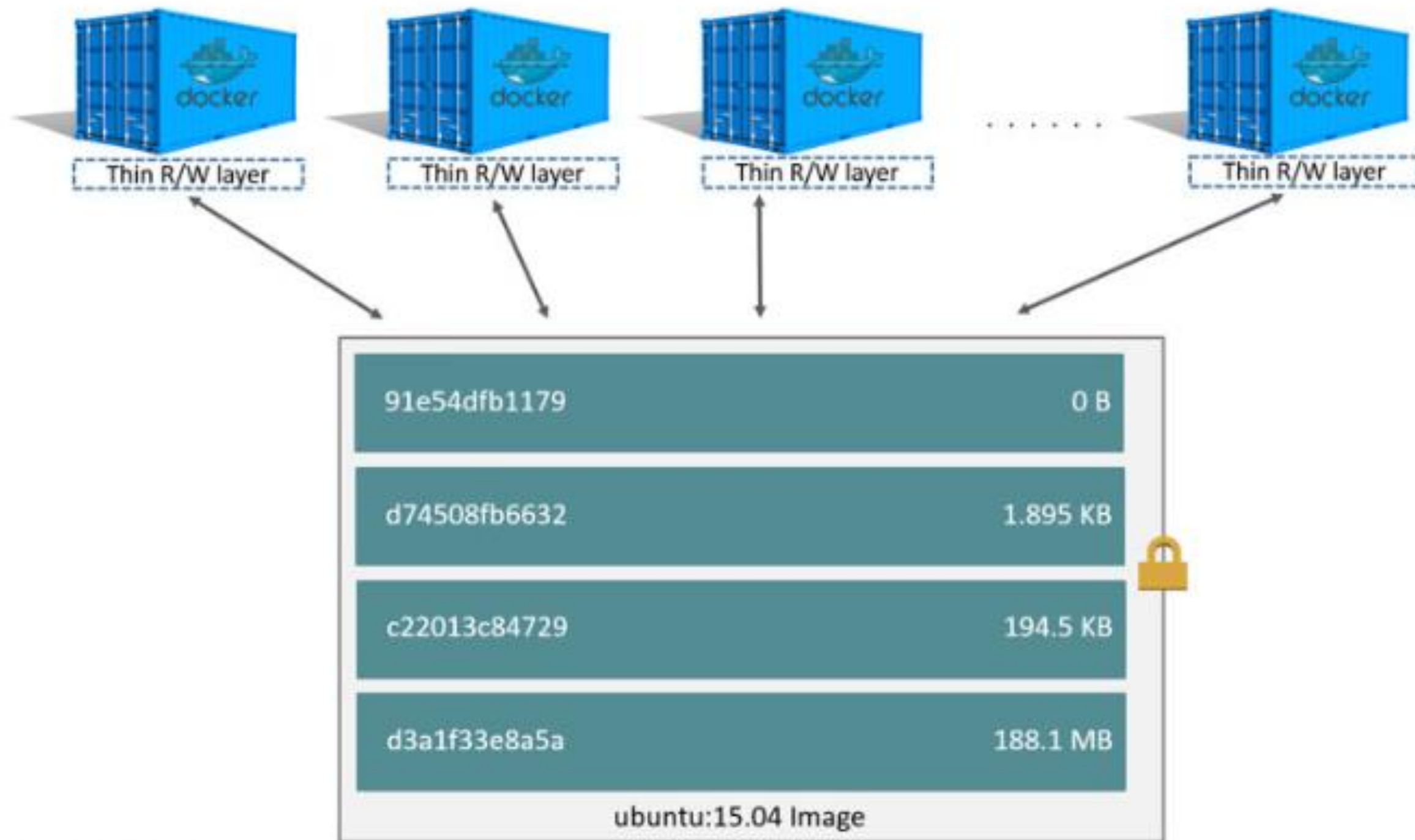
## Layers of Image

# Layers

A Docker image is built from a series of layers. Each layer is an instruction in the Dockerfile of the image. Except the very last layer, each layer is read-only.



# Layers



# Identifying the Layers

Layers of an image can be identified using the following commands:

```
$ docker images -a
```

Used to find the image ID

```
$ docker history --no-trunc <ImageID>
```

Used to find layers and image size

## Container Size on Disk

The user can use the **docker ps -s** command to view the approximate size of a running container. Two different columns which are related to the size of the container are:

- **Size:** the amount of data (on disk) that is used for the writable layer of each container.
- **Virtual size:** the amount of data used for the read-only image data used by the container plus the container's writable layer size.

# Container Size on Disk

The number of ways a container can take up disk space:

- Used for log files if the user uses the **json-file** logging driver
- Used for volumes and bind mounts which are used by the container
- Used for the container's configuration files, which are typically small
- Used for memory which are written to disk if swapping is enabled



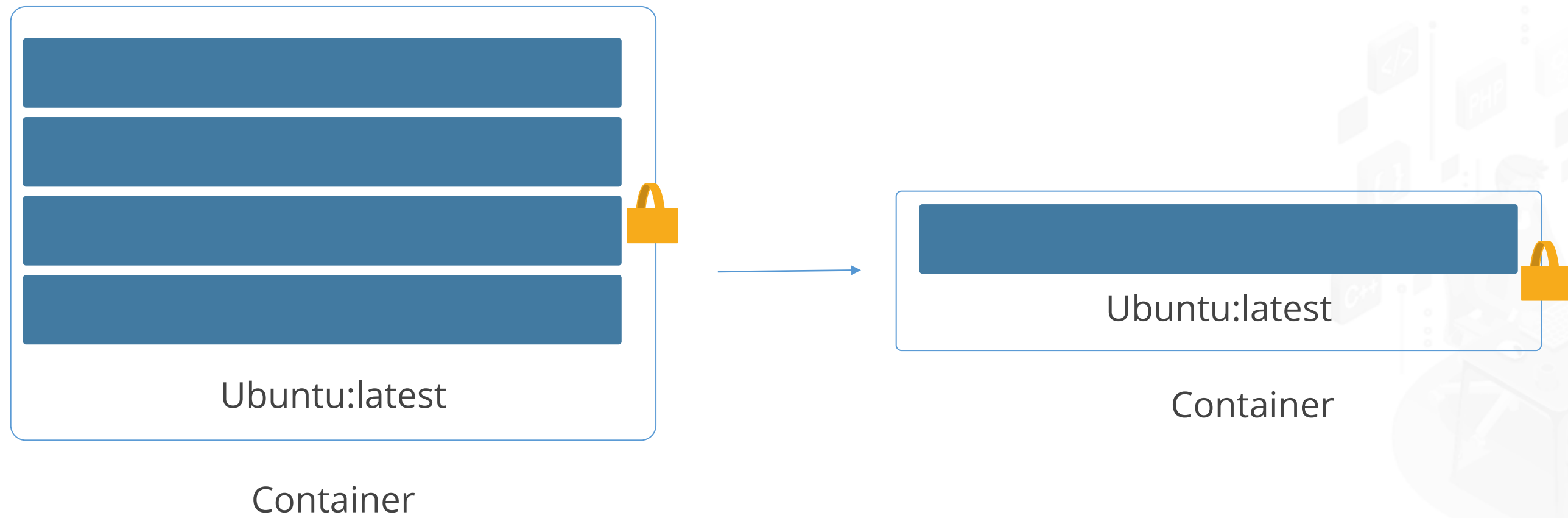
# FULL STACK

## Flattening



# Flattening

The purpose behind flattening the container is to reduce the size of the container.





# Flattening the Containers

Find the desired image by running *docker images*

1

Find the layers of the desired Docker image by running *docker image history <imageID>*

2

Create the container from the desired image

3

Export the container to a tar file

4

Import the container tar file

5

# FULL STACK

## Docker Commit

# Docker Commit

It commits the changes or settings of a container into a new image. By default, during the process of creating the commit, all the processes are paused as this helps in reduction of data corruption.

While committing, `--change` option is used to make changes to Dockerfile instructions such as `CMD`, `ENTRYPOINT`, `ENV`, `EXPOSE`, `LABEL`, `ONBUILD`, `USER`, `VOLUME`, and `WORKDIR`.

Committing a container:

```
$ docker ps
```

```
$ docker commit ContainerID repository:tag
```

```
$ docker images
```

# FULL STACK

## Tag an Image

# Tagging

Docker tags provide information about the image version or variant.

Tagging during building an image:

Command:

```
docker build -t repo_name:version_0.1 .
```

Current directory

Tag

Repository

Tag syntax

# Tagging

Tagging an image referenced by name:

Command:

```
docker image tag image_name: repository_name: version_0.1
```

Tag

Repository

Image name

Tag syntax

# Tagging

Tagging an image referenced by name and tag:

Command:

```
docker image tag image_name: image_tag repository_name: version_0.1
```

Tag

Repository

Image tag

Image name

Tag syntax



# Image Tagging



**Problem Statement:** You have been asked to tag an image and push it to a local registry or Docker Hub so that the images can be easily identified.

## Steps to Perform:

1. Tag an image using image ID.
2. Tag an image using image name.
3. Tag an image using image name and tag.
4. Tag an image for a private repository.

ASSISTED PRACTICE



# FULL STACK

## Filter and Format

# Filter

The format of filter flag is a key-value pair.

Filter option is used in *docker images* to filter:

- Images that are not tagged
- Images that are labelled
- Images by time
- Images by reference

# Filter

Filter option is used in *docker search* to filter:

- Images according to the stars scored
- Images according to their automation status
- Images according to their official status

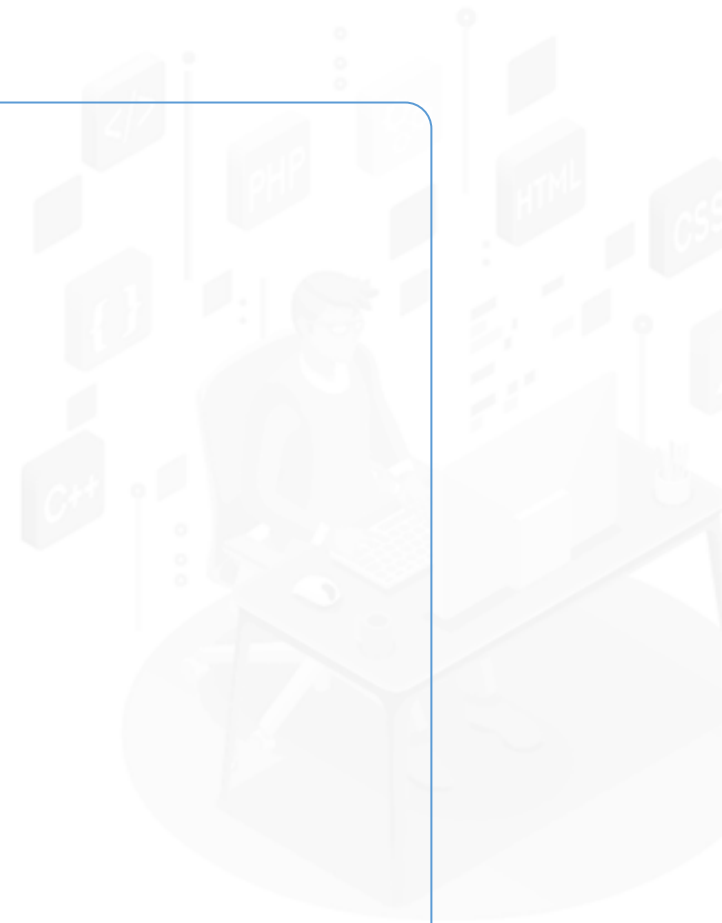


# Format

Format option is used in *docker image* to filter:

The format option *--format* helps in identifying:

- Image ID
- Image repository
- Image tag
- Image digest
- Image disk size
- Time at which the image was created
- Time elapsed since the creation of the image

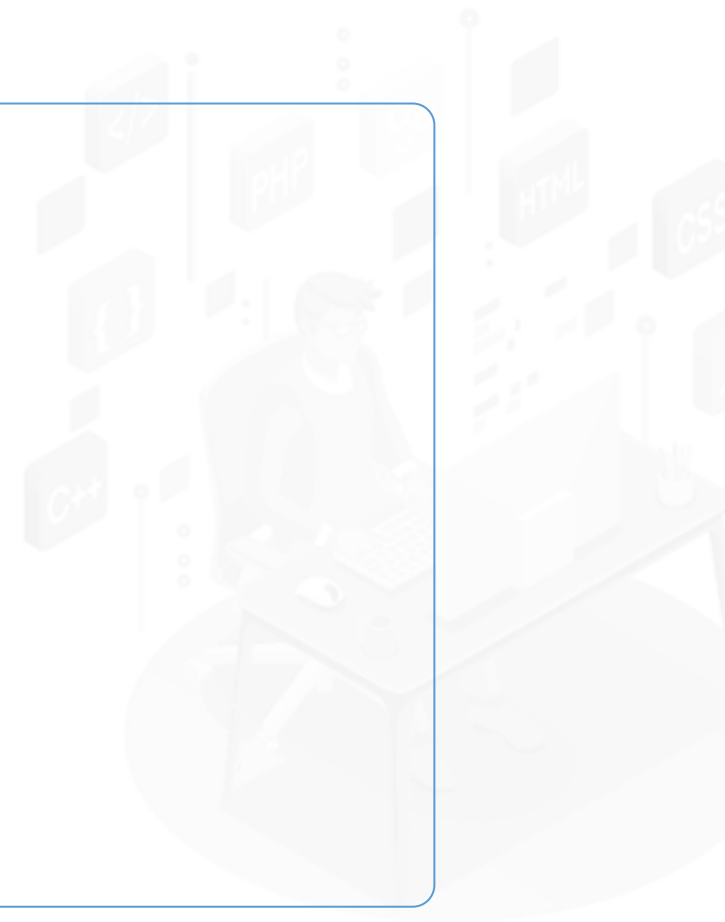


# Format

Format option is used in *docker search* to filter:

The format option `--format` helps in identifying:

- Image name
- Image description
- Image stars
- Official image
- Automated image



# FULL STACK

## Basic Commands

# Basic Commands

## Description:

## Command usage:

Listing the images

*docker image ls -a*

Option *-a* denotes list all the images

Remove images

*docker image rm -f*

Option *-f* denotes force removal

Remove unused images

*docker image prune -a*

Option *-a* denotes remove all the images

## The Copy-on-Write (COW) Strategy



# The Copy-on-Write (COW) Strategy

Copy-on-write is a strategy of sharing and copying files for maximum efficiency. If there is a file or directory within the image in a lower layer and another layer (including the writable layer) needs read access to it, it just uses the existing file.

When does a user use docker pull?

- To pull down an image from a repository, or create a container from an image that does not exist locally
- To pull down each layer separately and store in Docker's local storage area, which is usually **/var/lib/docker/** on Linux hosts

# The Copy-on-Write (COW) Strategy

If the user builds images from the two Docker files, the user can use **docker image** and **docker history** commands to verify that the cryptographic IDs of the shared layers are the same.

- 1 Make a new directory **cow-test/**.
- 2 Create a new file within **cow-test/**.
- 3 Copy the contents of the first Docker file into a new file called **Dockerfile.base**.
- 4 Copy the contents of the second Dockerfile into a new file called **Dockerfile**.
- 5 Build the first image within the **cow-test/** directory.
- 6 Build the second image.
- 7 Check out the size of the images.
- 8 Check out the layers that comprise each image.

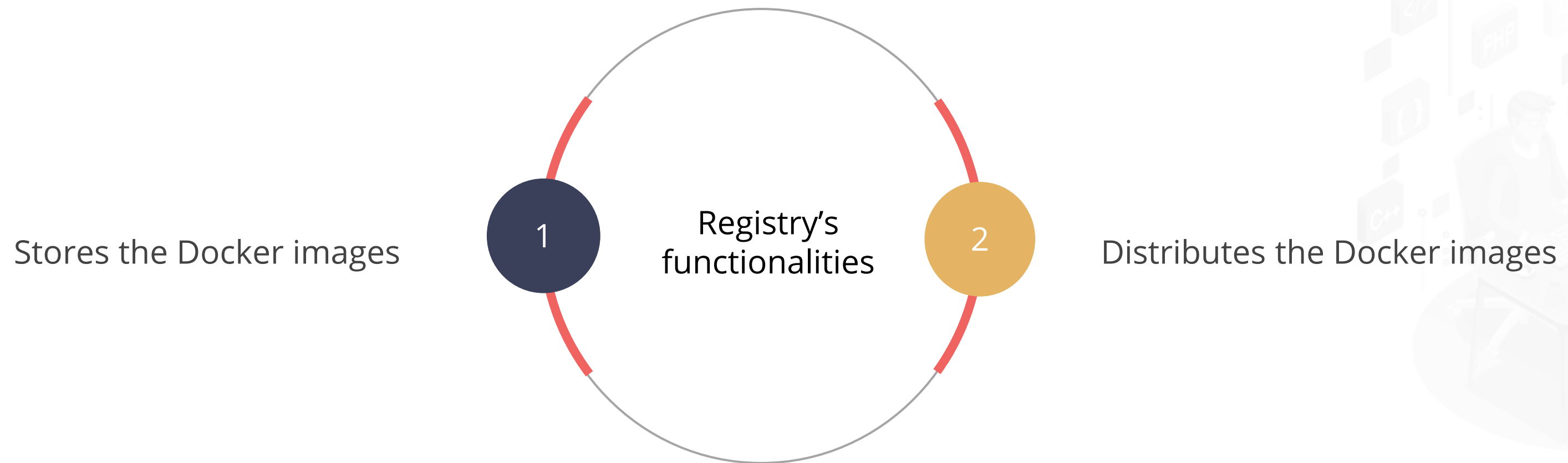


# FULL STACK

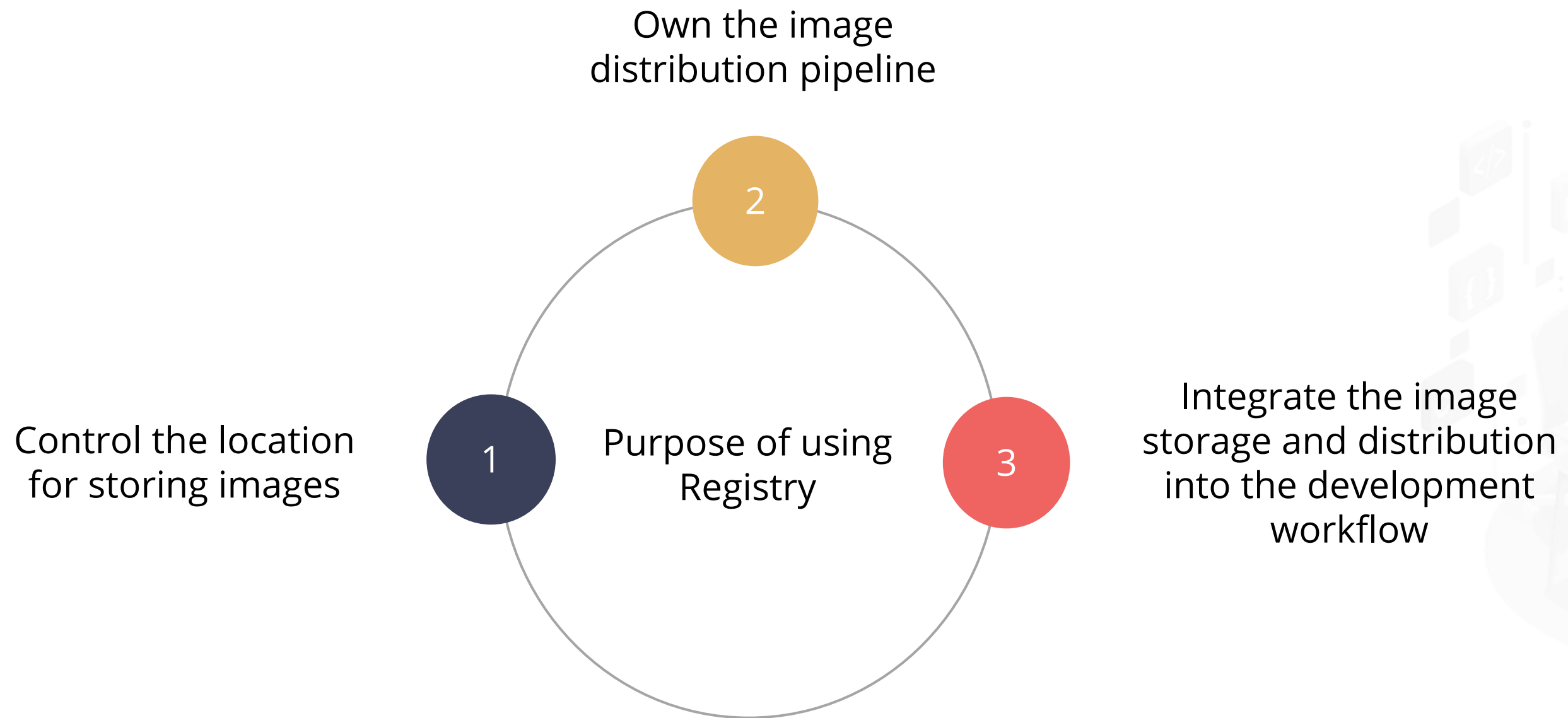
## Registry

# Registry

It is a stateless and scalable application that is compatible with the version 1.6.0 of Docker engine or higher.



# Registry



# Registry



# FULL STACK

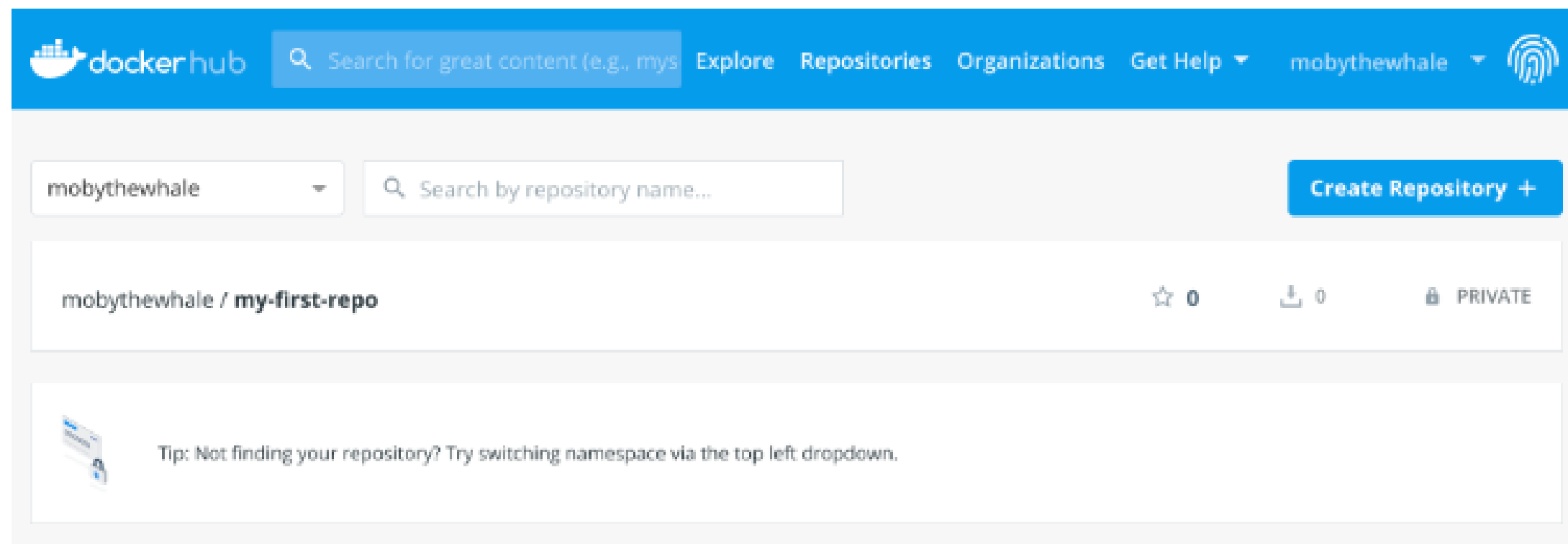
## Repositories

# Repositories

Docker Hub repositories allow the user to share container images with the team, customers, or the Docker community at large.

## Creating Repositories

1. Sign in to Docker Hub.
2. Click on Create Repository to create a repository.





# Repositories

The user can choose to put it in their Docker ID namespace

The repository name needs to be unique in that namespace, can be two to 255 characters, and can only contain lowercase letters, numbers, or "-" and "\_"

**When creating a new repository:**

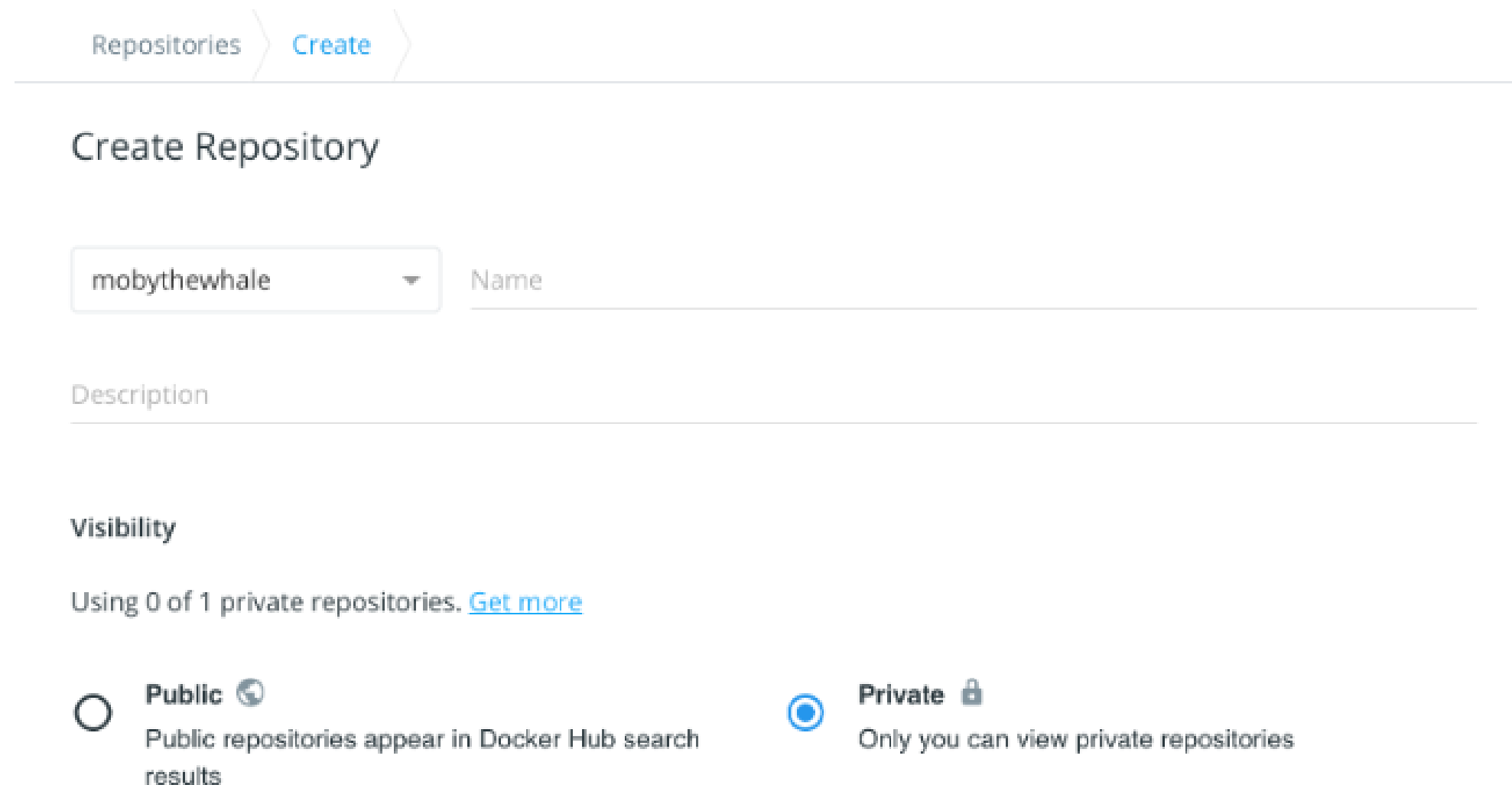
The description can be up to 100 characters and is used in the search result

The user can link a GitHub or Bitbucket account, or choose to do it later in the repository settings

# Private Repositories

Private repositories allow the user to keep container images private, either in their own account or within an organization or team.

To create a private repository, select **Private** when creating a repository:



Repositories > Create


## Create Repository


mobythewhale ▼ Name

Description

Visibility

Using 0 of 1 private repositories. [Get more](#)

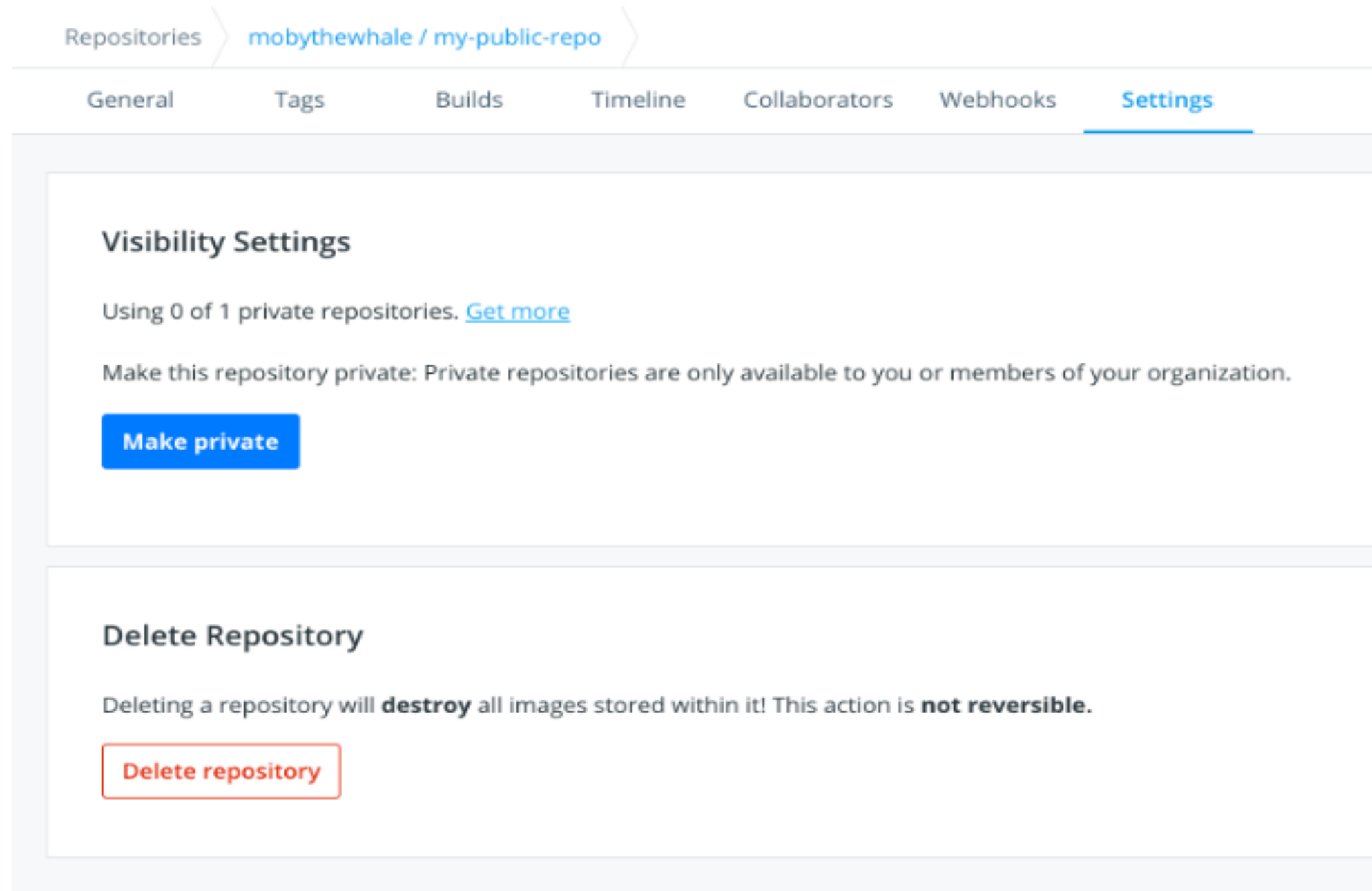
☐ **Public**  Public repositories appear in Docker Hub search results

☒ **Private**  Only you can view private repositories



# Private Repositories

The user can also make an existing repository private by going to its **Settings** tab:



# FULL STACK

## Docker Push and Pull

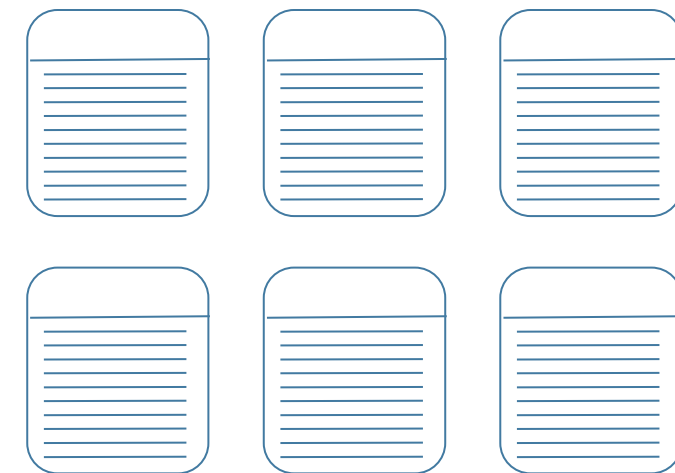
# Docker Push and Pull

Pushing the images to a registry makes them easily accessible to a larger population.



Image

Command: *docker push*



Multiple images on registry

# Docker Push and Pull

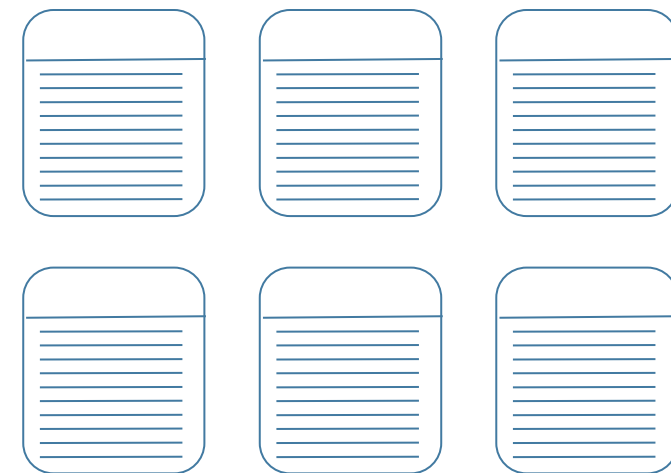
Pulling the images from a registry is done by using the following command:

```
docker pull [OPTIONS] NAME[:TAG | @DIGEST]
```



Image

Command: *docker pull*



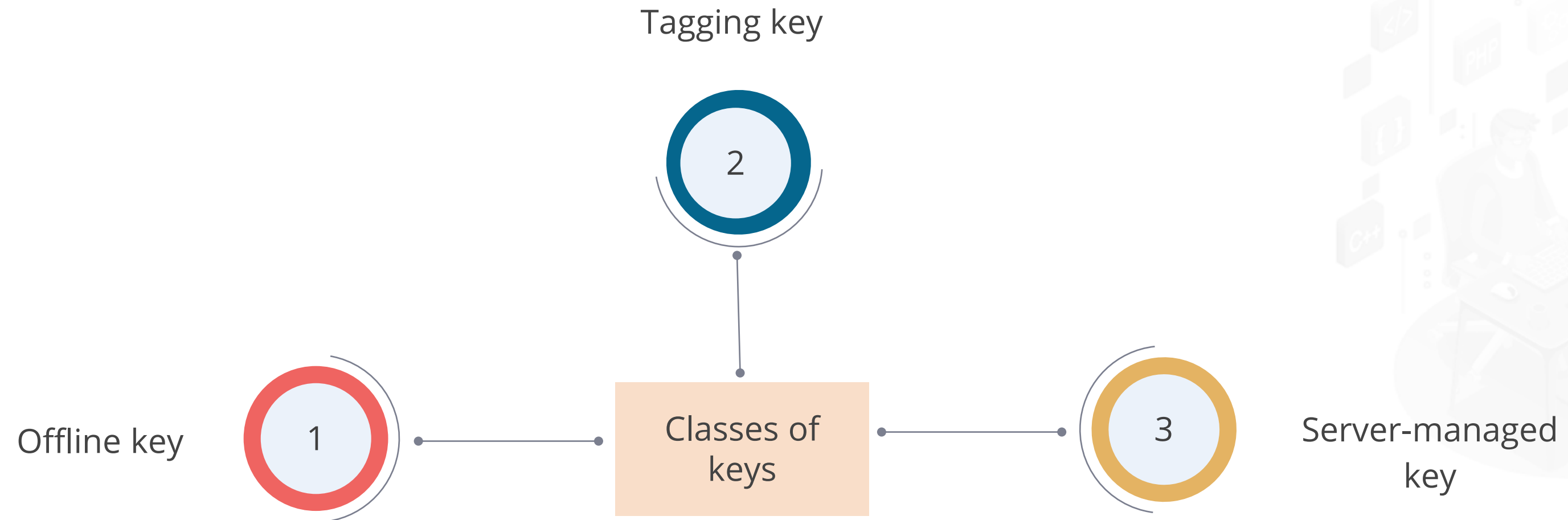
Multiple images on registry

# FULL STACK

## Content Trust

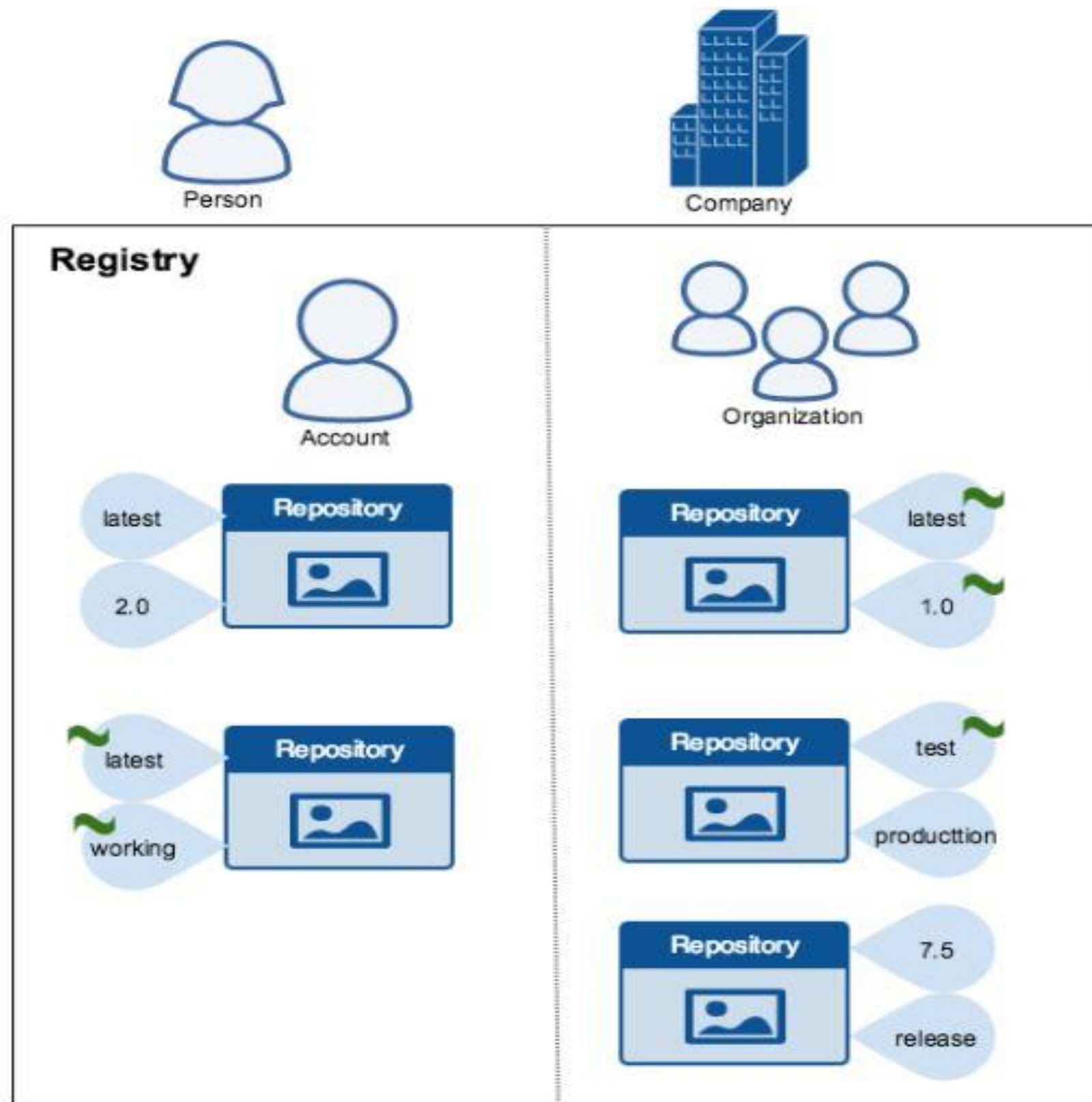
# Docker Content Trust

Docker Content Trust keys are used to manage the trust of an image tag.





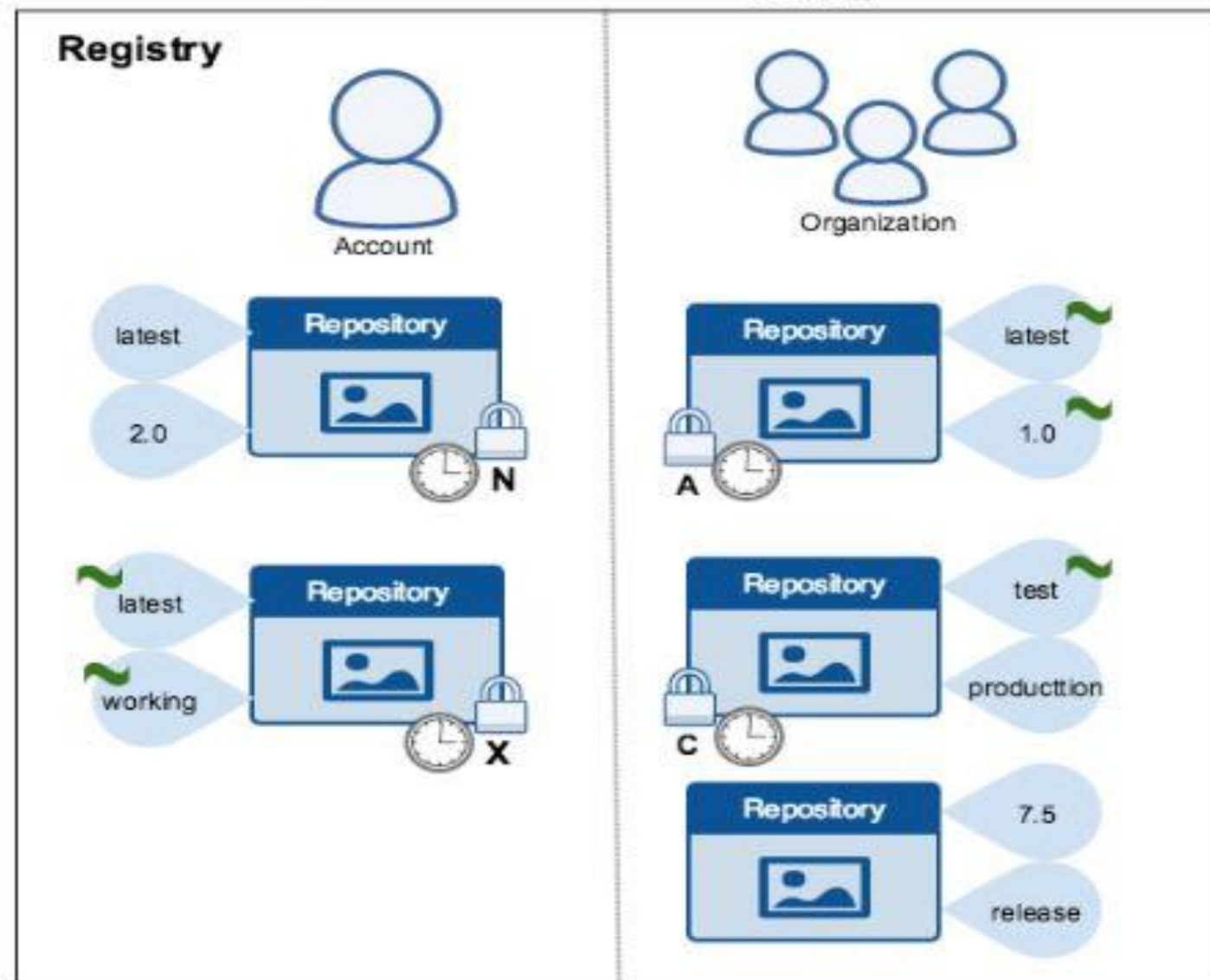
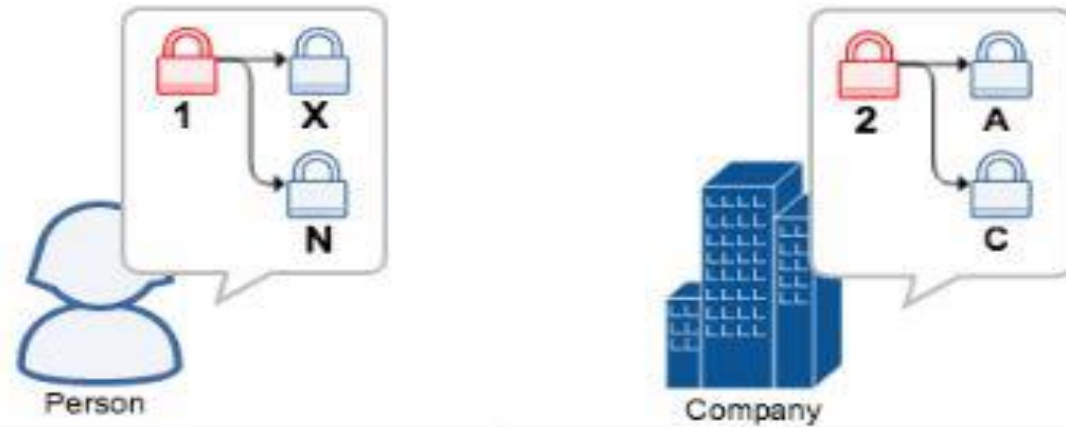
# Docker Content Trust



~ Signed tag



# Docker Content Trust



- It is used to create tagging keys. This key belongs to a person or organization.
- It resides with the client.
- It is associated with an image repository.
- Creators can push or pull any tag in the image repository by using a tagging key.
- This key resides with the client.
- Timestamp key is associated with an image repository.
- The key is created by docker.
- Timestamp key resides on the server.
- Signed tag.

# Deploy a Registry



**Problem Statement:** You have been asked by your team lead to deploy a registry to store images so that the images can be removed from the local cache.

## Steps to Perform:

1. Run a local registry.
2. Pull an image from Docker Hub to your registry and tag it.
3. Push the image to the local registry and remove it from local cache.
4. Pull the image from the local registry and later stop the registry.

ASSISTED PRACTICE

# Configure a Registry



**Problem Statement:** Your manager has asked you to configure a registry by overriding its configuration file or environment variables.

## Steps to Perform:

1. Configure the *rootdirectory* of the *filesystem* storage backend.
2. Specify a *configuration variable* from the environment by passing *-e* argument in the *docker run* command.
3. Override the entire *configuration file* by creating a new file named *config.yml*.

ASSISTED PRACTICE



# Log in to a Registry



**Problem Statement:** Your team lead has asked you to login to a Docker registry and store the login credentials in the *config* file to keep them safe.

## Steps to Perform:

1. Login to a local Docker registry.
2. Use the *login* command non-interactively by setting *--password-stdin* flag.
3. Specify the credential's store in *\$HOME/.docker/config.json* to let the Docker engine use it.
4. Use the *store* command to take the JSON payload from the *STDIN*.
5. Use the *GET* command to write a JSON payload to *STDOUT*.

ASSISTED PRACTICE

# Push an Image to Docker Hub



**Problem Statement:** You have been asked by your supervisor to push an image to Docker Hub so that it can be publicly accessed.

## Steps to Perform:

1. Creating a Docker image from the Dockerfile.
2. Use *docker login* command to login to your Docker Hub account.
3. Tag the Docker image and push the image to your Docker Hub repository.
4. Go to your Docker Hub account and navigate to *Repositories* to see your recently pushed image.

ASSISTED PRACTICE

# Push an Image to a Registry



**Problem Statement:** Your colleague requests you to share a Docker image that you have created. You must push the Docker image to a shared registry so that he can use it.

## Steps to Perform:

1. Pull an image from Docker Hub.
2. Tag the image you want to push.
3. Push the image to the local registry.
4. Stop the registry once your work is done.

ASSISTED PRACTICE

# FULL STACK

## Prune Images and Containers



# Cleaning of Images

The approach of cleaning up unused objects in Docker is referred to as garbage collection. Such objects are images, containers, volumes, and networks.

- Docker usually doesn't remove the objects unless requested by the user.
- Docker provides a prune command.
- The **docker system prune** helps to clean up multiple types of objects at once.
- The **docker system prune** command is a shortcut that prunes images, containers, and networks.

# Prune Images

The **docker image prune** command allows you to clean up unused images.

The command **docker image prune** only cleans up dangling images.

A dangling image is one that is not tagged and is not referenced by any container.

The user can use the following command to remove dangling images:

```
$ docker image prune
```

# Prune Images

## Options:

| Name, shorthand | Description                                       |
|-----------------|---|
| --all , -a      | Removes all unused images, not just dangling ones |
| --filter        | Provides filter values (e.g. 'until=')            |
| --force , -f    | Does not prompt for confirmation                  |

## Parent Command

| Command    | Description                                       |
|------------|---|
| --all , -a | Removes all unused images, not just dangling ones |



# Prune Containers

When the user stops a container, it is not automatically removed. To see all containers on the Docker host, including stopped containers, use **docker ps -a**.

The user can use the following command to remove unused containers:

```
$ docker container prune
```

# Inspect, Remove, and Prune Images



**Problem Statement:** Your manager has requested you to inspect and remove an image if it is not being used. The manager also wants you to prune all dangling images to free up any unwanted space.

## Steps to Perform:

1. Pull an image from *Docker Hub*.
2. Inspect the image for details such as *ID and ContainerConfig*.
3. List the Docker images and copy the image ID of the image to be removed.
4. Remove the image using remove command. Use *--force* flag if required.
5. Use the *prune* command to remove all unused or dangling images.

ASSISTED PRACTICE

# Pull and Delete an Image



**Problem Statement:** You are working on a project and require a base image that can be pulled from Docker Hub. You must also delete the image once its purpose is complete.

## Steps to Perform:

1. Pull the *ubuntu:14.04* image from Dockerhub or using the *Digest*.
2. Tag the image so that it points to your registry.
3. Push the image to your local registry and delete the local-cache.
4. Pull the image from your registry and use it wherever needed.
5. List all the images and delete the *untagged* or *dangling* images.

ASSISTED PRACTICE

## Key Takeaways

- Images are created by Dockerfile where each instruction on the Dockerfile adds a layer to the image.
- Images are flattened first by creating a container from the desired image. The container is then exported to a tarball and import back.
- Docker Hub requires zero maintenance and provides a free-to-use and hosted Registry.
- Docker push helps store the images to the registry and docker pull helps access those images.
- Docker Content Trust tags the images with a digital signature to identify the integrity of the images.

