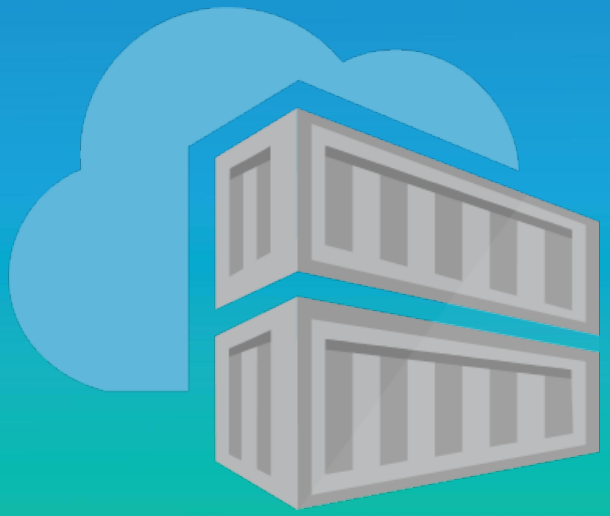
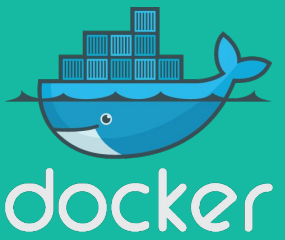
Docker Components





Docker can build images automatically by reading the instructions from a Dockerfile.
A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.
Dockerfile:Dockerfile is a simple text file that consists of instructions to build Docker images.
Mentioned below is the syntax of a Dockerfile:
Syntax
comments
command argument argument1...
Example
Print "Get Certified. Get Ahead"
Run echo "Get Certified. Get Ahead"
Now, let's have a look at how to build a Docker image using a dockerfile.

Dockerfile

```dockerfile
# Create image based on the official python 3.8 image from dockerhub
FROM python:3.8

# Create directory where our app will be placed
RUN mkdir -p /usr/src/app

# Change directory so that our command run from this new directory
WORKDIR /usr/src/app

# Copy dependency and app code
COPY . /usr/src/app

# Install required dependency
RUN pip3 install -r requirements.txt

# Provide the instruction to start application or Entrypoint to start the application
ENTRYPOINT ["python"]

#Specifies what command to run within the container.
CMD ["app.py"]
```

docker

Dockerfile

```dockerfile
# Create image based on the official node 14 image from dockerhub
FROM node:14

# Create directory where our app will be placed
RUN mkdir -p /usr/src/app

# Change directory so that our command run from this new directory
WORKDIR /usr/src/app

# Copy dependency and app code
COPY package*.json app.js ./

# Install dependencies
RUN npm install

#Expose the sport the app runs in
EXPOSE 3000

# Provide the instruction to start application or Entrypoint to start the application
ENTRYPOINT ["node"]

#Specifies what command to run within the container.
#CMD [ "node","app.js"]
CMD [ "app.js"]
```

Dockerfile

```
# Create image based on the official openjdk 8 image from dockerhub
FROM openjdk:8-jdk-alpine

#Bind volume for
VOLUME /tmp

# Read Argument from maven
ARG JAR_FILE

# Copy dependency and app code to app.jar
COPY ${JAR_FILE} app.jar

# Provide the instruction to start application or Entrypoint to start the application
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]

#COPY target/docker-message-server-1.0.0.jar message-server-1.0.0.jar
```

Environment variables are supported by the following list of instructions in the `Dockerfile` :

- `ADD`
- `COPY`
- `ENV`
- `EXPOSE`
- `FROM`
- `LABEL`
- `STOPSIGNAL`
- `USER`
- `VOLUME`
- `WORKDIR`
- `ONBUILD` (when combined with one of the supported instructions above)

.dockerignore file

Before the docker CLI sends the context to the docker daemon, it looks for a file named `.dockerignore` in the root directory of the context. If this file exists, the CLI modifies the context to exclude files and directories that match patterns in it. This helps to avoid unnecessarily sending large or sensitive files and directories to the daemon and potentially adding them to images using `ADD` or `COPY`.

The CLI interprets the `.dockerignore` file as a newline-separated list of patterns similar to the file globs of Unix shells. For the purposes of matching, the root of the context is considered to be both the working and the root directory. For example, the patterns `/foo/bar` and `foo/bar` both exclude a file or directory named `bar` in the `foo` subdirectory of `PATH` or in the root of the git repository located at `URL`. Neither excludes anything else.

If a line in `.dockerignore` file starts with `#` in column 1, then this line is considered as a comment and is ignored before interpreted by the CLI.

Here is an example `.dockerignore` file:

```
# comment
*/temp*
*/*/temp*
temp?
```

This file causes the following build behavior:

This file causes the following build behavior:

Rule	Behavior
`# comment`	Ignored.
`*/temp*`	Exclude files and directories whose names start with `temp` in any immediate subdirectory of the root. For example, the plain file `/somedir/temporary.txt` is excluded, as is the directory `/somedir/temp`.
`*/*/temp*`	Exclude files and directories starting with `temp` from any subdirectory that is two levels below the root. For example, `/somedir/subdir/temporary.txt` is excluded.
`temp?`	Exclude files and directories in the root directory whose names are a one-character extension of `temp`. For example, `/tempa` and `/tempb` are excluded.

FROM

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

Or

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

Or

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

The `FROM` instruction initializes a new build stage and sets the *Base Image* for subsequent instructions. As such, a valid `Dockerfile` must start with a `FROM` instruction. The image can be any valid image – it is especially easy to start by **pulling an image** from the *Public Repositories*.

Understand how ARG and FROM interact

`FROM` instructions support variables that are declared by any `ARG` instructions that occur before the first `FROM`.

```
ARG  CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD  /code/run-app

FROM extras:${CODE_VERSION}
CMD  /code/run-extras
```

An `ARG` declared before a `FROM` is outside of a build stage, so it can't be used in any instruction after a `FROM`. To use the default value of an `ARG` declared before the first `FROM` use an `ARG` instruction without a value inside of a build stage:

```
ARG VERSION=latest
FROM busybox:$VERSION
ARG VERSION
RUN echo $VERSION > image_version
```

RUN

RUN has 2 forms:

- `RUN <command>` (*shell* form, the command is run in a shell, which by default is `/bin/sh -c` on Linux or `cmd /S /C` on Windows)
- `RUN ["executable", "param1", "param2"]` (*exec* form)

The `RUN` instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the `Dockerfile`.

Layering `RUN` instructions and generating commits conforms to the core concepts of Docker where commits are cheap and containers can be created from any point in an image's history, much like source control.

The *exec* form makes it possible to avoid shell string munging, and to `RUN` commands using a base image that does not contain the specified shell executable.

The default shell for the *shell* form can be changed using the `SHELL` command.

In the *shell* form you can use a `\` (backslash) to continue a single RUN instruction onto the next line. For example, consider these two lines:

```
RUN /bin/bash -c 'source $HOME/.bashrc; \
echo $HOME'
```

Together they are equivalent to this single line:

```
RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'
```

To use a different shell, other than '/bin/sh', use the *exec* form passing in the desired shell. For example:

```
RUN ["/bin/bash", "-c", "echo hello"]
```

CMD

The `CMD` instruction has three forms:

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

There can only be one `CMD` instruction in a `Dockerfile`. If you list more than one `CMD` then only the last `CMD` will take effect.

The main purpose of a `CMD` is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an `ENTRYPOINT` instruction as well.

If `CMD` is used to provide default arguments for the `ENTRYPOINT` instruction, both the `CMD` and `ENTRYPOINT` instructions should be specified with the JSON array format.

> **Note**
>
> The *exec* form is parsed as a JSON array, which means that you must use double-quotes (") around words not single-quotes (').

Unlike the *shell* form, the *exec* form does not invoke a command shell. This means that normal shell processing does not happen. For example, `CMD [ "echo", "$HOME" ]` will not do variable substitution on `$HOME`. If you want shell processing then either use the *shell* form or execute a shell directly, for example: `CMD [ "sh", "-c", "echo $HOME" ]`. When using the exec form and executing a shell directly, as in the case for the shell form, it is the shell that is doing the environment variable expansion, not docker.

When used in the shell or exec formats, the `CMD` instruction sets the command to be executed when running the image.

If you use the *shell* form of the `CMD`, then the `<command>` will execute in `/bin/sh -c`:

```
FROM ubuntu
CMD echo "This is a test." | wc -
```

If you want to **run your `<command>` without a shell** then you must express the command as a JSON array and give the full path to the executable. **This array form is the preferred format of `CMD`.** Any additional parameters must be individually expressed as strings in the array:

```
FROM ubuntu
CMD ["/usr/bin/wc","--help"]
```

If you would like your container to run the same executable every time, then you should consider using `ENTRYPOINT` in combination with `CMD`. See *ENTRYPOINT*.

If the user specifies arguments to `docker run` then they will override the default specified in `CMD`.

LABEL

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

The `LABEL` 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. A few usage examples:

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

An image can have more than one label. You can specify multiple labels on a single line. Prior to Docker 1.10, this decreased the size of the final image, but this is no longer the case. You may still choose to specify multiple labels in a single instruction, in one of the following two ways:

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

```
LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"
```

Labels included in base or parent images (images in the `FROM` line) are inherited by your image. If a label already exists but with a different value, the most-recently-applied value overrides any previously-set value.

To view an image's labels, use the `docker image inspect` command. You can use the `--format` option to show just the labels;

```
$ docker image inspect --format='' myimage
```

```
{
  "com.example.vendor": "ACME Incorporated",
  "com.example.label-with-value": "foo",
  "version": "1.0",
  "description": "This text illustrates that label-values can span multiple lines.",
  "multi.label1": "value1",
  "multi.label2": "value2",
  "other": "value3"
}
```

MAINTAINER (deprecated)

```
MAINTAINER <name>
```

The `MAINTAINER` instruction sets the *Author* field of the generated images. The `LABEL` instruction is a much more flexible version of this and you should use it instead, as it enables setting any metadata you require, and can be viewed easily, for example with `docker inspect`. To set a label corresponding to the `MAINTAINER` field you could use:

```
LABEL org.opencontainers.image.authors="SvenDowideit@home.org.au"
```

This will then be visible from `docker inspect` with the other labels.

EXPOSE

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

The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the 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, or the `-P` flag to publish all exposed ports and map them to high-order ports.

By default, `EXPOSE` assumes TCP. You can also specify UDP:

```
EXPOSE 80/udp
```

To expose on both TCP and UDP, include two lines:

```
EXPOSE 80/tcp
EXPOSE 80/udp
```

In this case, if you use `-P` with `docker run`, the port will be exposed once for TCP and once for UDP. Remember that `-P` uses an ephemeral high-ordered host port on the host, so the port will not be the same for TCP and UDP.

Regardless of the `EXPOSE` settings, you can override them at runtime by using the `-p` flag. For example

```
$ docker run -p 80:80/tcp -p 80:80/udp ...
```

To set up port redirection on the host system, see using the -P flag. The `docker network` command supports creating networks for communication among containers without the need to expose or publish specific ports, because the containers connected to the network can communicate with each other over any port. For detailed information, see the overview of this feature.

ENV

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

The `ENV` instruction sets the environment variable `<key>` to the value `<value>`. This value will be in the environment for all subsequent instructions in the build stage and can be replaced inline in many as well. The value will be interpreted for other environment variables, so quote characters will be removed if they are not escaped. Like command line parsing, quotes and backslashes can be used to include spaces within values.

Example:

```
ENV MY_NAME="John Doe"
ENV MY_DOG=Rex\ The\ Dog
ENV MY_CAT=fluffy
```

The `ENV` instruction allows for multiple `<key>=<value> ...` variables to be set at one time, and the example below will yield the same net results in the final image:

```
ENV MY_NAME="John Doe" MY_DOG=Rex\ The\ Dog \
    MY_CAT=fluffy
```

The environment variables set using `ENV` will persist when a container is run from the resulting image. You can view the values using `docker inspect`, and change them using `docker run --env <key>=<value>`.

Environment variable persistence can cause unexpected side effects. For example, setting `ENV DEBIAN_FRONTEND=noninteractive` changes the behavior of `apt-get`, and may confuse users of your image.

If an environment variable is only needed during build, and not in the final image, consider setting a value for a single command instead:

```
RUN DEBIAN_FRONTEND=noninteractive apt-get update && apt-get install -y ...
```

Or using `ARG`, which is not persisted in the final image:

```
ARG DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y ...
```

ADD

ADD has two forms:

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

The latter form is required for paths containing whitespace.

> **Note**
>
> The `--chown` feature is only supported on Dockerfiles used to build Linux containers, and will not work on Windows containers. Since user and group ownership concepts do not translate between Linux and Windows, the use of `/etc/passwd` and `/etc/group` for translating user and group names to IDs restricts this feature to only be viable for Linux OS-based containers.

The `ADD` instruction copies new files, directories or remote file URLs from `<src>` and adds them to the filesystem of the image at the path `<dest>`.

Multiple `<src>` resources may be specified but if they are files or directories, their paths are interpreted as relative to the source of the context of the build.

Each `<src>` may contain wildcards and matching will be done using Go's filepath.Match rules. For example:

To add all files starting with "hom":

```
ADD hom* /mydir/
```

In the example below, `?` is replaced with any single character, e.g., "home.txt".

```
ADD hom?.txt /mydir/
```

The `<dest>` is an absolute path, or a path relative to `WORKDIR`, into which the source will be copied inside the destination container.

The example below uses a relative path, and adds "test.txt" to `<WORKDIR>/relativeDir/`:

```
ADD test.txt relativeDir/
```

COPY

COPY has two forms:

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

This latter form is required for paths containing whitespace

> **Note**
>
> The `--chown` feature is only supported on Dockerfiles used to build Linux containers, and will not work on Windows containers. Since user and group ownership concepts do not translate between Linux and Windows, the use of `/etc/passwd` and `/etc/group` for translating user and group names to IDs restricts this feature to only be viable for Linux OS-based containers.

The `COPY` instruction copies new files or directories from `<src>` and adds them to the filesystem of the container at the path `<dest>`.

Multiple `<src>` resources may be specified but the paths of files and directories will be interpreted as relative to the source of the context of the build.

Each `<src>` may contain wildcards and matching will be done using Go's filepath.Match rules. For example:

To add all files starting with "hom":

```
COPY hom* /mydir/
```

In the example below, `?` is replaced with any single character, e.g., "home.txt".

```
COPY hom?.txt /mydir/
```

The `<dest>` is an absolute path, or a path relative to `WORKDIR`, into which the source will be copied inside the destination container.

The example below uses a relative path, and adds "test.txt" to `<WORKDIR>/relativeDir/`:

```
COPY test.txt relativeDir/
```

Whereas this example uses an absolute path, and adds "test.txt" to `/absoluteDir/`

```
COPY test.txt /absoluteDir/
```

ENTRYPOINT

ENTRYPOINT has two forms:

The *exec* form, which is the preferred form:

```
ENTRYPOINT ["executable", "param1", "param2"]
```

The *shell* form:

```
ENTRYPOINT command param1 param2
```

An `ENTRYPOINT` allows you to configure a container that will run as an executable.

For example, the following starts nginx with its default content, listening on port 80:

```
$ docker run -i -t --rm -p 80:80 nginx
```

Command line arguments to `docker run <image>` will be appended after all elements in an *exec* form `ENTRYPOINT`, and will override all elements specified using `CMD`. This allows arguments to be passed to the entry point, i.e., `docker run <image> -d` will pass the `-d` argument to the entry point. You can override the `ENTRYPOINT` instruction using the `docker run --entrypoint` flag.

The *shell* form prevents any `CMD` or `run` command line arguments from being used, but has the disadvantage that your `ENTRYPOINT` will be started as a subcommand of `/bin/sh -c`, which does not pass signals. This means that the executable will not be the container's `PID 1` - and will *not* receive Unix signals - so your executable will not receive a `SIGTERM` from `docker stop <container>`.

Only the last `ENTRYPOINT` instruction in the `Dockerfile` will have an effect.

VOLUME

```
VOLUME ["/data"]
```

The `VOLUME` instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers. The value can be a JSON array, `VOLUME ["/var/log/"]`, or a plain string with multiple arguments, such as `VOLUME /var/log` or `VOLUME /var/log /var/db`. For more information/examples and mounting instructions via the Docker client, refer to *Share Directories via Volumes* documentation.

The `docker run` command initializes the newly created volume with any data that exists at the specified location within the base image. For example, consider the following Dockerfile snippet:

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

This Dockerfile results in an image that causes `docker run` to create a new mount point at `/myvol` and copy the `greeting` file into the newly created volume.

Notes about specifying volumes

Keep the following things in mind about volumes in the `Dockerfile`.

- **Volumes on Windows-based containers**: When using Windows-based containers, the destination of a volume inside the container must be one of:
  - a non-existing or empty directory
  - a drive other than `C:`
- **Changing the volume from within the Dockerfile**: If any build steps change the data within the volume after it has been declared, those changes will be discarded.
- **JSON formatting**: The list is parsed as a JSON array. You must enclose words with double quotes (`"`) rather than single quotes (`'`).
- **The host directory is declared at container run-time**: The host directory (the mountpoint) is, by its nature, host-dependent. This is to preserve image portability, since a given host directory can't be guaranteed to be available on all hosts. For this reason, you can't mount a host directory from within the Dockerfile. The `VOLUME` instruction does not support specifying a `host-dir` parameter. You must specify the mountpoint when you create or run the container.

USER

```
USER <user>[:<group>]
```

or

```
USER <UID>[:<GID>]
```

The `USER` instruction sets the user name (or UID) and optionally the user group (or GID) to use when running the image and for any `RUN`, `CMD` and `ENTRYPOINT` instructions that follow it in the `Dockerfile`.

Note that when specifying a group for the user, the user will have *only* the specified group membership. Any other configured group memberships will be ignored.

Warning

When the user doesn't have a primary group then the image (or the next instructions) will be run with the `root` group.

On Windows, the user must be created first if it's not a built-in account. This can be done with the `net user` command called as part of a Dockerfile.

```
FROM microsoft/windowsservercore
# Create Windows user in the container
RUN net user /add patrick
# Set it for subsequent commands
USER patrick
```

WORKDIR

```
WORKDIR /path/to/workdir
```

The `WORKDIR` instruction sets the working directory for any `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD` instructions that follow it in the `Dockerfile`. If the `WORKDIR` doesn't exist, it will be created even if it's not used in any subsequent `Dockerfile` instruction.

The `WORKDIR` instruction can be used multiple times in a `Dockerfile`. If a relative path is provided, it will be relative to the path of the previous `WORKDIR` instruction. For example:

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

The output of the final `pwd` command in this `Dockerfile` would be `/a/b/c`.

The `WORKDIR` instruction can resolve environment variables previously set using `ENV`. You can only use environment variables explicitly set in the `Dockerfile`. For example:

```
ENV DIRPATH=/path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
```

The output of the final `pwd` command in this `Dockerfile` would be `/path/$DIRNAME`

If not specified, the default working directory is `/`. In practice, if you aren't building a Dockerfile from scratch (`FROM scratch`), the `WORKDIR` may likely be set by the base image you're using.

Therefore, to avoid unintended operations in unknown directories, it is best practice to set your `WORKDIR` explicitly.

ARG

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

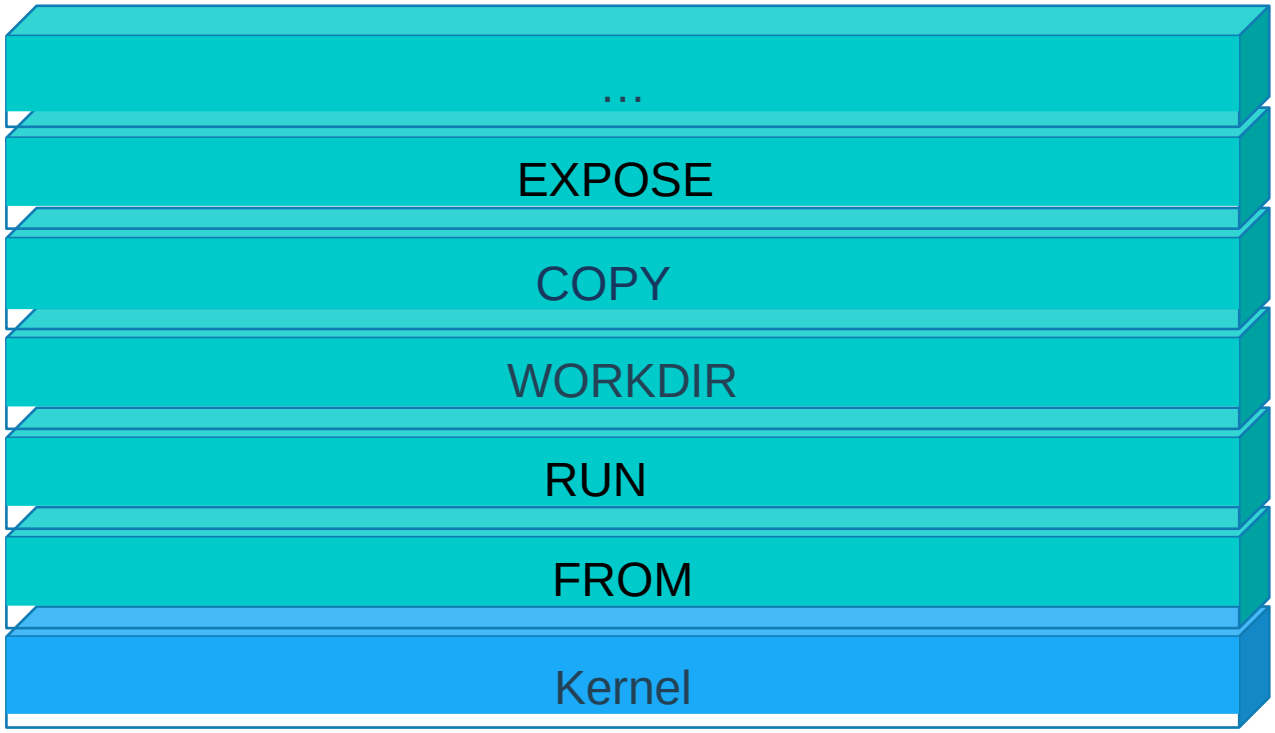
The `ARG` instruction defines a variable that users can pass at build-time to the builder with the `docker build` command using the `--build-arg <varname>=<value>` flag. If a user specifies a build argument that was not defined in the Dockerfile, the build outputs a warning.

```
[Warning] One or more build-args [foo] were not consumed.
```

A Dockerfile may include one or more `ARG` instructions. For example, the following is a valid Dockerfile:

```
FROM busybox
ARG user1
ARG buildno
# ...
```

Each Dockerfile Command Creates a Layer



Section 2:

Anatomy of a Docker Container
Docker Volumes
Volume Use Cases

Docker Image Pull: Pulls Layers

```
Alexander@DESKTOP-90ATKET MINGW64 ~/Docker/Demo
$ docker pull nginx:latest
latest: Pulling from library/nginx
bc95e04b23c0: Pull complete
f3186e650f4e: Pull complete
9ac7d6621708: Pull complete
Digest: sha256:b81f317384d7388708a498555c28a7cce778a8f291d90021208b3eba3fe74887
Status: Downloaded newer image for nginx:latest
```

Docker Volumes

- Volumes mount a directory on the host into the container at a specific location
- Can be used to share (and persist) data between containers
  - Directory persists after the container is deleted
    - Unless you explicitly delete it
- Can be created in a Dockerfile or via CLI

Why Use Volumes

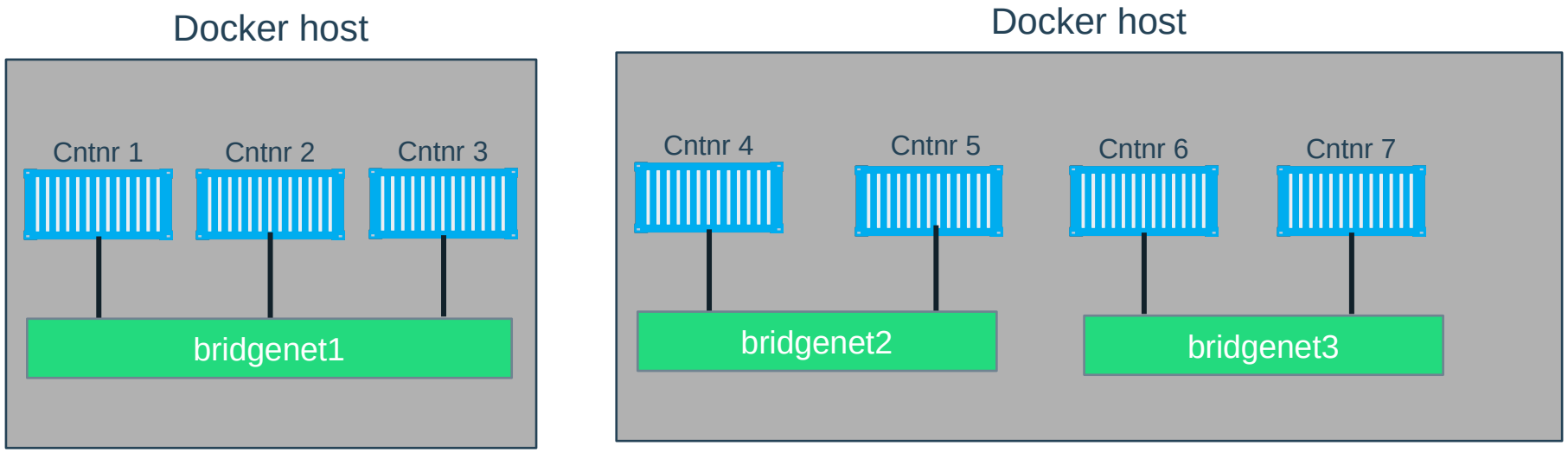
Mount local source code into a running container

```
docker container run -v $(pwd):/usr/src/app/  myapp
```

- Improve performance
  - As directory structures get complicated traversing the tree can slow system performance
- Data persistence

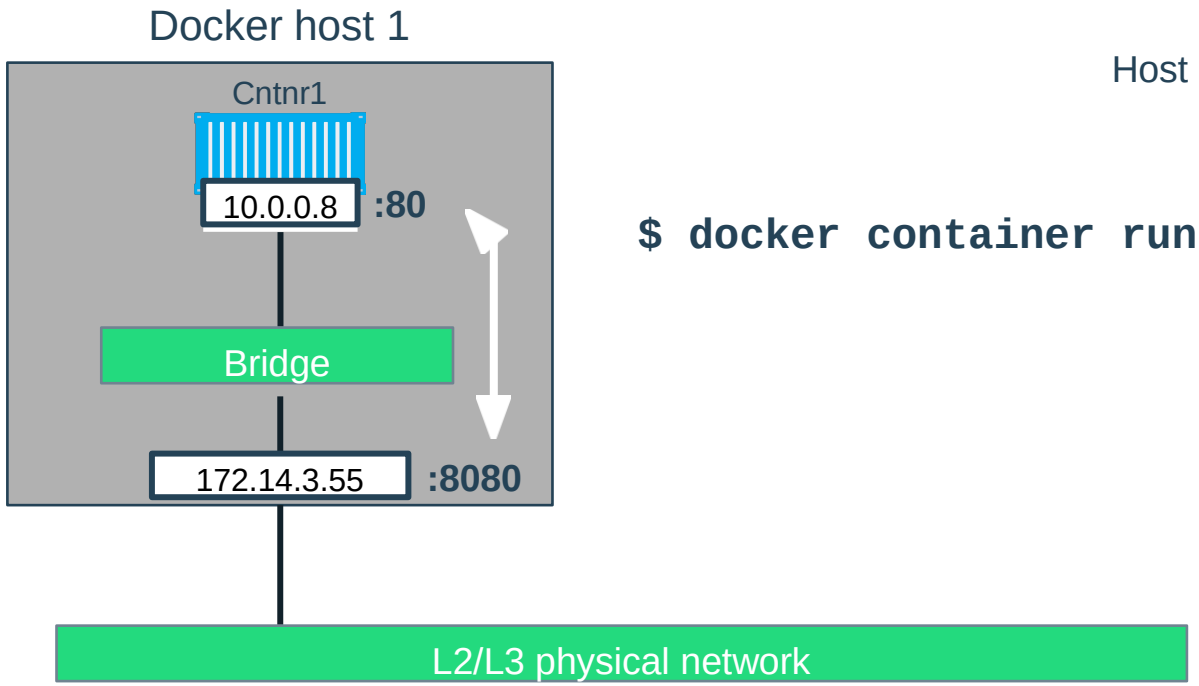
Section 3:
Networking

What is Docker Bridge Networking




```
docker network create -d bridge --name
bridgenet1
```

Docker Bridge Networking and Port Mapping



Host port

Container port

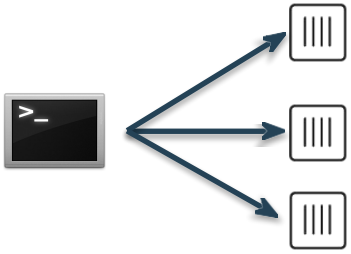
```
$ docker container run -p 8080:80 ...
```


docker

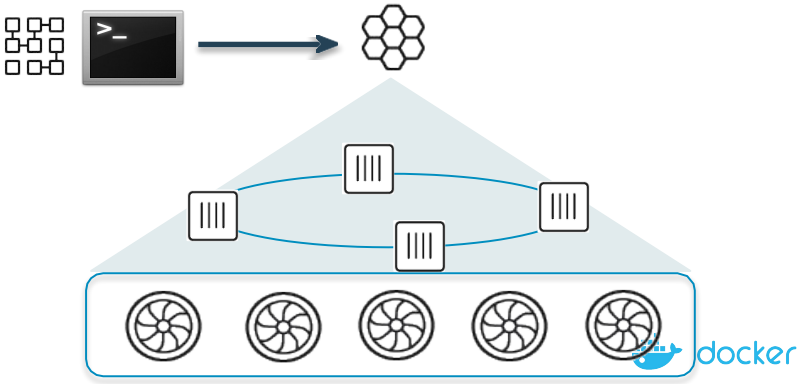
Section 4:
Docker Compose

Docker Compose: Multi Container Applications

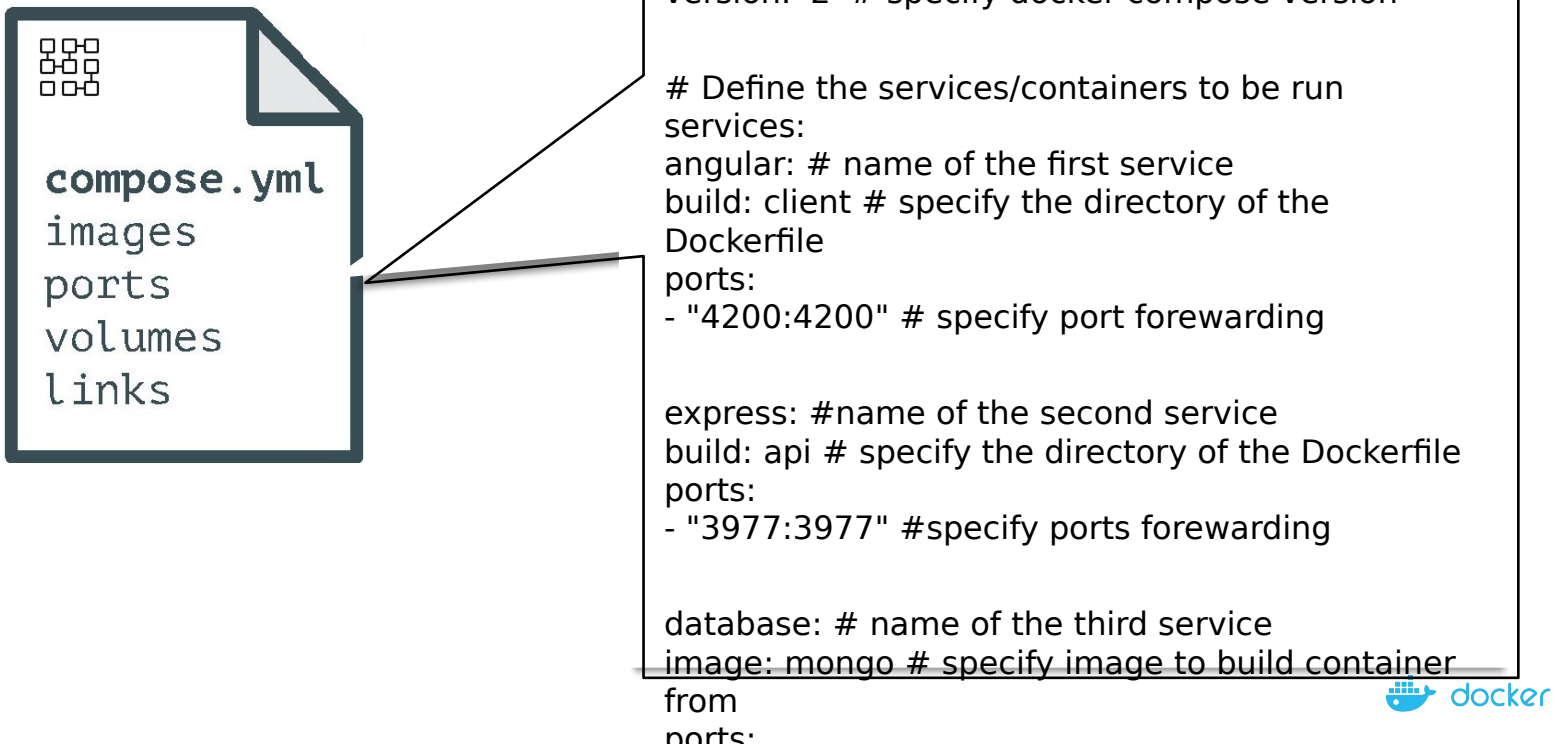
Build and run one container at a time
Manually connect containers together
Must be careful with dependencies and start up order



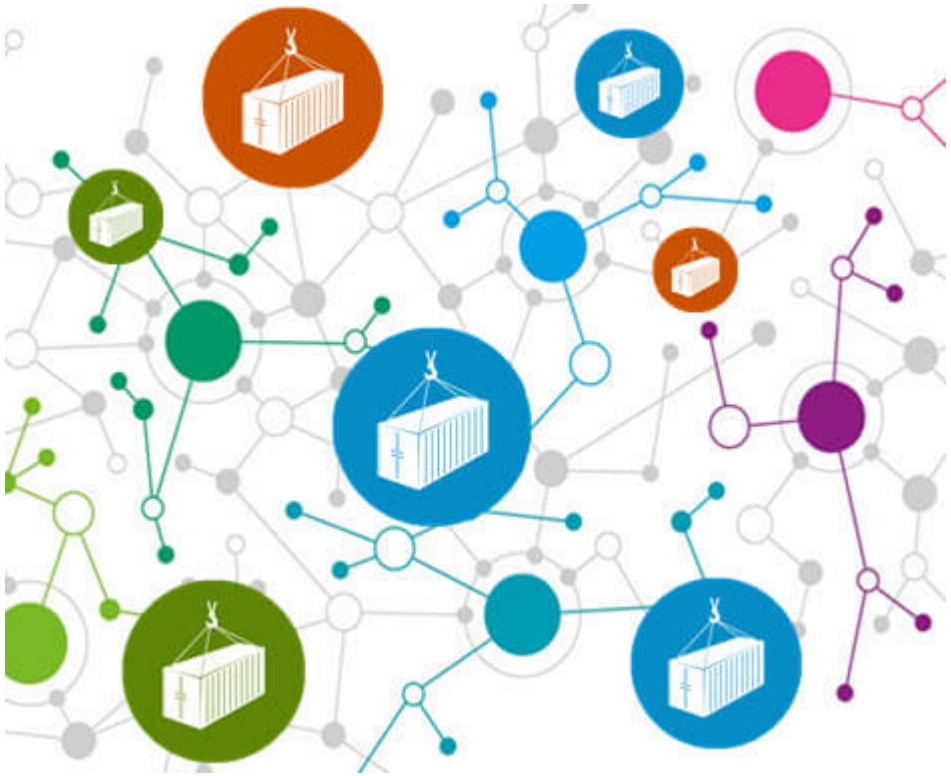
Define multi container app in compose.yml file
Single command to deploy entire app
Handles container dependencies
Works with Docker Swarm, Networking, Volumes, Universal Control Plane



Docker Compose: Multi Container Applications

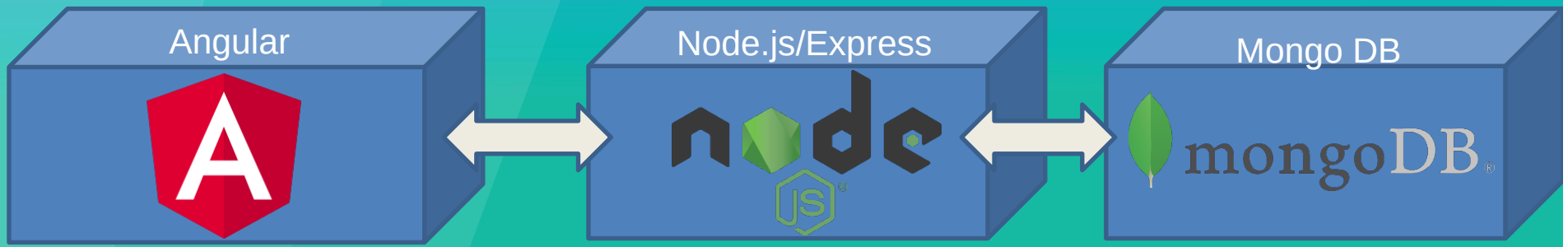


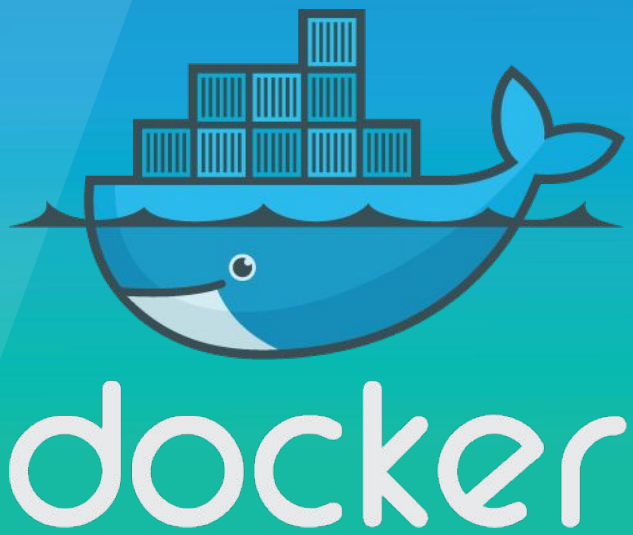
Docker Compose: Scale Container Applications



docker

Demo




docker