Container Virtualization with Docker

CS Tools, Tips and Tricks Seminar @ McGill University

Marton Bur

2019 Winter

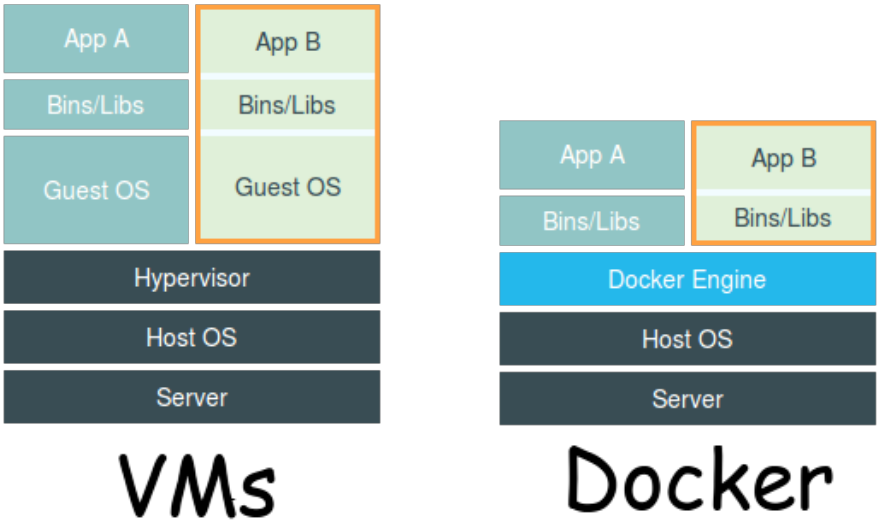
Table of Contents

PDF version

1. Overview of Virtualization

Using a virtualization method and building an image provides consistent environments. The virtualization method can be categorized based on how it mimics hardware to a guest operating system and emulates guest operating environment. Primarily, there are two main types of virtualization:

- **Emulation and paravirtualization** is based on some type of hypervisor which is responsible for translating guest OS kernel code to software instructions or directly executes it on the bare-metal hardware. A virtual machine provides a computing environment with dedicated resources, requests for CPU, memory, hard disk, network and other hardware resources that are managed by a virtualization layer. A virtual machine has its own OS and full software stack.
- **Container-based virtualization**, also known as operating system-level virtualization, enables multiple isolated executions within a single operating system kernel. It has the best possible performance and density, while featuring dynamic resource management. The isolated virtual execution environment provided by this type of virtualization is called a *container* and can be viewed as a well-defined group of processes.



2. Docker foundations

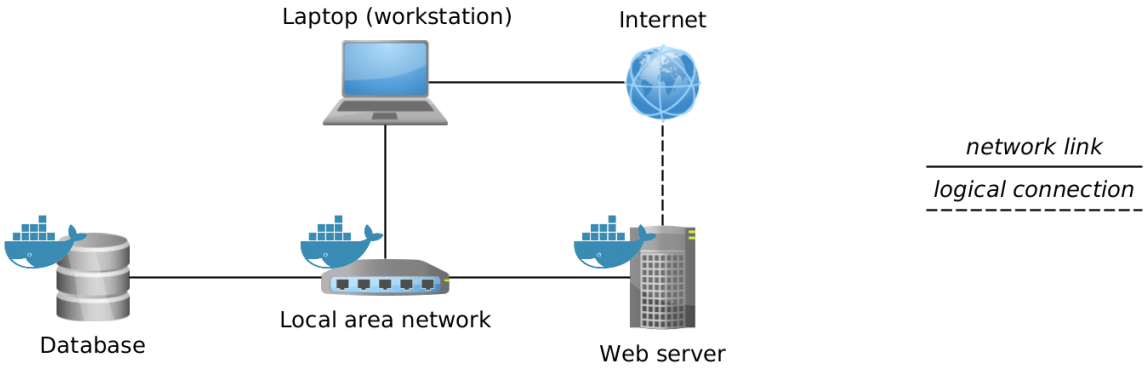
The core concepts of Docker are **images** and **containers**.

A **Docker image** is a read-only template with instructions for creating a Docker container. For example, an image might contain an Ubuntu operating system with an Apache web server and your web application installed. You can build or update images from scratch or download and use images created by others. An image may be based on, or may extend, one or more other images. A Docker image is described in text file called a Dockerfile, which has a simple, well-defined syntax.

A **Docker container** is a runnable instance of a Docker image. You can run, start, stop, move, or delete a container using Docker API or CLI commands. When you run a container, you can provide

configuration metadata such as networking information or environment variables.

Exercise. We are going to create a simple virtual network on which a web server runs a *Java Spring application*. The functionality provided by the web server via a RESTful API is to (i) add people and (ii) add events to the system, as well as to (iii) register selected people to selected events. This example application is adapted from the ECSE321 course tutorial. We are also exposing this functionality to the Internet. Furthermore, we run a Postgres Database that is also connected to this same network. This database is used by the java application running on the web server to store the registration info. An architecture diagram of the system is shown below.



NOTE For any Docker command, you can use the `--help` command line switch to learn about that particular command and its possible parameters. For example, try `docker run --help`

2.1. Docker containers

2.1.1. Running a container

To create and run a container, one needs to specify an image on which the container is based on. Luckily, docker has the support for downloading images automatically from an online *repository* in which it identifies images by their names and versions.

Things to try:

- `docker run hello-world`
- `docker run --rm busybox ping www.google.com`
- `docker run -it busybox`

Exercise. Setting up the database can be done by issuing `docker run --name postgresql-server -e POSTGRES_PASSWORD=pass -e POSTGRES_USER=user -e POSTGRES_DB=eventregistration -d postgres` command.

NOTE	Later in this tutorial, we will take a quick look at DockerHub where images like *postgres* are hosted and their settings are documented.

Once a container is started *and it is running*, `docker exec` can be used to execute a command within that container. For example, list what files are in the current working folder of the DB server with `docker exec postgresql-server ls`.

2.1.2. Container management

Docker offers commands (among many) to list, stop, start, and remove containers. Furthermore, the `docker inspect` command can tell several details about the configuration of the given container.

Things to try:

- `docker ps`
- `docker ps --all` — example output:

```
CONTAINER ID  IMAGE                 COMMAND                  STATUS
9bba8a2a3f81  makisyu/texlive-2016  "/bin/bash -c 'sleep…"  Exited (0) 4 days ago
cd005b9af0af  makisyu/texlive-2016  "/bin/bash -c 'sleep…"  Exited (0) 4 days ago
b92dd4d5886d  eclipse/che           "/scripts/entrypoint…"  Exited (2) 5 days ago
```

- `docker rm <CONTAINER_ID>`
- `docker container prune`
- `docker inspect <CONTAINER_ID>`

IMPORTANT	Once a container is removed (deleted), data stored within the container is lost unless additional steps are taken.

2.2. Docker images

2.2.1. Creating a new image

An image is defined in a **Dockerfile**. Every image starts from a base image, e.g. from `ubuntu`, a base Ubuntu image. The Docker image is built from the base image using a simple, descriptive set of steps we call instructions, which are stored in a `Dockerfile`. Main dockerfile **instructions**: * `FROM`: specifies an already existing image that is used as a starting point when creating a new image * `RUN`: executes a command during build * `COPY`: copies a file to the image * `VOLUME`: mounts a volume to the image * `CMD`: default command that is executed once a container is started from the image — a Dockerfile can have only one of this! * `WORKDIR`: specifies the default working directory

NOTE	There are commands with the same functionality for a running container. For example, the `COPY` instruction for an image is complemented with `docker cp`. For an already running container, `docker cp` can copy a file to the container's filesystem (or the other way around).

The `docker build .` command builds an **image** from a **Dockerfile and a context**. The build's **context** is the set of files at a specified location PATH or URL (in this case the current directory, `.`). The PATH is a directory on your local filesystem. The URL is a Git repository location. Add the `-f` switch to specify the Dockerfile location, if it is not present in the root context.

> *Exercise.* Use the `java` image to create a new image for the Spring Web application using `docker build` (name this new image *example-spring-app*). You need to copy both the *example-webapp.jar* and the *application.properties* to the image. The web server should be started by `java -jar example-webapp.jar` when the container starts.

2.2.2. Managing images

Docker provides similar commands to the ones available for containers, one just needs to add the `image` keyword to the command. For example, `docker images ls --all` yields

```
REPOSITORY            IMAGE ID      CREATED       SIZE
eclipse/che           8956a46aa7e3  10 days ago   51.3MB
gradle                e7f185032db8  2 months ago  820MB
busybox               6ad733544a63  3 months ago  1.13MB
makisyu/texlive-2016  bb92f3e57f6b  9 months ago  5.42GB
```

For more details, see the Dockerfile reference.

2.2.3. DockerHub

DockerHub is a place where images can be uploaded and shared. You can download images from DockerHub with `docker pull`. Once registered and executed `docker login`, the `docker push` can be used to publish your images.

3. Docker networking

3.1. Virtual LAN

You can create an isolated virtual local network for your containers with `docker network create` command. This Docker blog posts explains networks in details, but for us a thorough discussion is out of scope.

> *Exercise.* Use the `docker inspect` command to verify that the web server and the database server containers are on the same network.

3.2. Expose containerized services to the Internet

In addition to managing virtual networks for containers, docker can automatically manages

firewall rules for setting up port forwarding your containers' services and the physical network interface of the host. Two steps need to be done to achieve this:

1. **A port should be exposed.** A port may be exposed by default for a container if the image was built using the `EXPOSE <PORT_NUMBER>` instruction. For individual containers, the `--expose` switch can be used with `docker run`.
2. **A post should be published.** When starting a new container, the `docker run` command must be specified which ports to expose with the `-p` switch so that it can automatically configure the corresponding firewall rules of the host. This will also ensure that the rules are removed once the container is removed.

> *Exercise.* Start/remove and start the web server with exposing and publishing port 8080 on the host's port of your choice. Observe what extra rules were added to the host's firewall with `sudo iptables -t nat -L -n`.

4. Saving your work

There are multiple approaches to save your work when dealing with Docker. This section overviews three common methods and illustrates them with small exercises using the *busybox* image.

4.1. Creating a new image from a container

Once you have already started a container, you can save its state to a new image (with a different name than what was used to start up the original container) by issuing `docker commit`. This allows you to start up a new container from the committed state any number of times and continue your work from a saved state.

> *Exercise.* Start a new instance of busybox with `docker run -it --name savetest busybox`. Create a file *greetings.txt* and add the `"Hello world!"` content to it. Use the `docker commit savetest busybox-greeting` command to save a new image named *busybox-greeting*. Create and run a new container from *busybox-greeting* and see if the created file is really there.

4.2. Saving and loading an image

Once you have created an image, one of the simplest way of moving it across hosts is to use `docker save` and `docker load`. It is important to remember that only images can be loaded/saved, so that one needs to commit first if there is data inside the container that needs to be saved. Docker can save/load `.tar` files.

> *Exercise.* Save the *busybox-greeting* image to *busybox-greeting.tar*. Then, try re-loading the image.

4.3. Saving data by mounting a volume from the host

In cases when you would like to provide input to/save input from a program that is running inside a container, the `docker cp` command may be inconvenient to use for handling several multiple input/output files. An alternative solution is to mount a folder from the hosts' filesystem as a volume in the container and use that folder to share files between the host and the container.

> *Exercise.* Mount a volume to a container created from the *busybox* image and try updating the folder content from both the host and the container. Investigate how to use the `-v <HOST_FOLDER>:<CONTAINER_FOLDER>` switch of `docker run`!

5. Miscellaneous

5.1. Running docker without sudo

Follow the steps in this answer on askubuntu, namely:

1. Add the docker group if it doesn't already exist: `sudo groupadd docker`
2. Add the connected user `$USER` to the docker group. Change the user name to match your preferred user if you do not want to use your current user: `sudo gpasswd -a $USER docker`
3. Either do a `newgrp docker` or log out/in to activate the changes to groups.

You can use `docker run hello-world` to check if you can run docker without sudo.