



UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386



Tools 2026 – Container Platforms

Shigehiko Schamoni
21.05.2026

With material from F. Beenen (SuS)

Overview

1 Introduction

- What are Containers?
- Container Platforms

2 Using Docker

- Basics
- Commands
- Working with Containers
- Advanced Usage
- Security

3 Using Apptainer

- Apptainer Features
- Basic Commands
- Using Apptainer

4 Exercise

What are Containers?

- Containers provide an isolated environment to execute programs.
- Containers isolate the application and its runtime environment from the infrastructure.
- Very similar to Virtual Machines (VMs): You can run multiple virtual operating systems on a host that are separated from each other.
- Yet different than VMs: A container shares parts of the *operating system* with the host but may have its own separated network, compute resources, storage and execution environment.

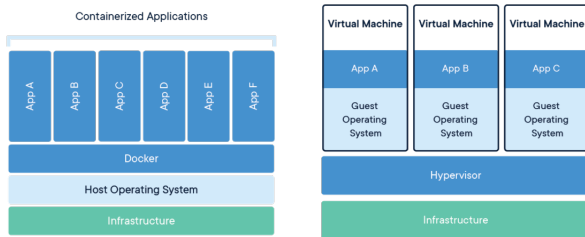


Figure: <https://www.docker.com/resources/what-container/>

Why use Containers?

- Containers can be set up and torn down much faster than VMs.
 - One application = one container!
 - Example: Compiler (gcc) in Container. For each compilation job, launch a Container, let it do its thing, tear down Container.
- Containers are typically stateless: A “clean” execution environment is created upon start and the Container should not have any side effects.
- Containers are portable: They bundle everything they need so they can be moved/executed on another host without issues (like VMs – with the same restrictions).
- Dependencies of programs can be managed in a more fine-grained way:
 - App A needs a library with version 1.2
 - App B requires version 1.3 of this library.
 - Libraries are not spread across system folders but are inside the Container!

Applications of Containers

- As "VM Lite"
 - Long-living Containers that contain independent services.
 - Examples: Database-Servers, Webservers, Load-balancers.
 - Same approach as deploying VMs but with less overhead.
- Microservice-based applications made out of multiple Containers
 - Applications are developed in a distributed fashion.
 - Applications use numerous micro-services that can be scaled out individually.
- As "job environment" for scheduled tasks
 - Used by build tools like Jekyll, GitLab, ...
 - Task: Create Container, execute a script in a defined environment, deliver an artifact, destroy Container.

Container Platforms

1 Docker

- The industry standard.
- <https://www.docker.com/>

2 Apptainer (formerly Singularity)

- Aims to overcome the problems and limitation of Docker.
- <https://apptainer.org/>

Similar but different: application containers

- AppImage
- Flatpack: backed by RedHat
- Snap: backed by Canonical (Ubuntu)

What is Docker

- Docker is a platform/software to run Containers!
- Docker is used to manage Containers:
 - I Start/Stop Container
 - Access Container (e. g. run a shell)
 - Attach networking to Containers
 - Manage storage shares between host and Containers
- Docker provides means to define/create the "working environment" of a Container (å this is called an Image).

What are Images?

- An Image is the bundle of software and configurations that are used as base for a Container.
- Container vs. Image?
 - A Container is the instance of an Image!
 - Multiple Containers may be started from the same Image and are independent.
- What makes up an Image is defined in a *Dockerfile*.
 - The Dockerfile can be regarded as a script (or definition) to compile the Image.
 - If the Dockerfile is built, the Image can be used to start Containers.
 - You can share the Image with others via a Registry (e.g. <https://hub.docker.com/>).

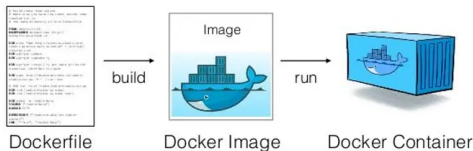


Figure: <https://nilesh93.medium.com/practical-guide-on-writing-a-dockerfile-for-your-application-89376f88b3b5>

Find Images on the Registry

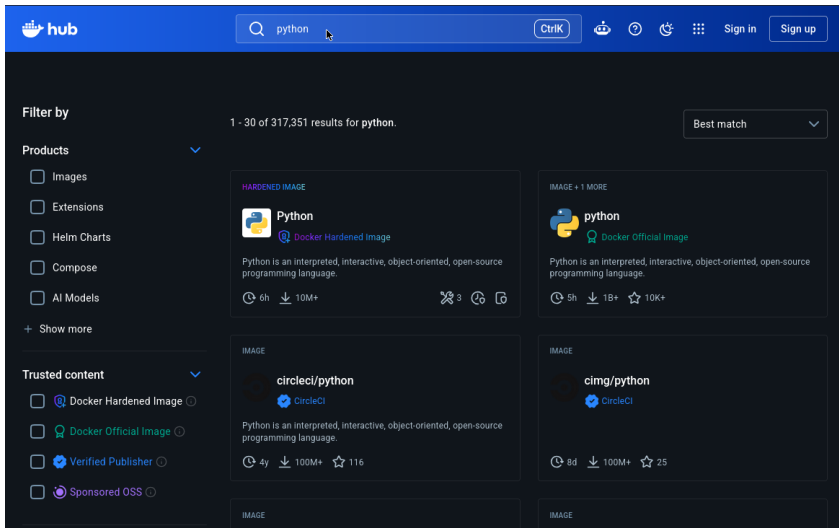


Figure: <https://hub.docker.com/search>

The Docker Daemon

- Docker is managed by the Docker daemon (as a system service).
- You can interact with the daemon by using the docker command.
- To do this, you need very high privileges (i. e. you need to be root effectively...).
- Basic commands
 - `docker images` : Shows Images currently available on your machine
 - `docker ps` : Shows running Containers

Starting a Container (I)

- Docker Containers are meant to be shipped with everything they need to execute...
- Our first Container launch will be a "hello world" example.
- For this purpose, the hello-world Image exists on the Hub.
- We run the Container by typing `docker run hello-world`

Starting a Container (II)

```
user@computer:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
4f55086f7dd0: Pull complete
Digest: sha256:0e760fdabc48ba8041e7c6db999bb40bfca508b4be580ac75d32c4e29d202ce
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
`$ docker run -it ubuntu bash`

Share images, automate workflows, and more with a free Docker ID:
<https://hub.docker.com/>

Starting Ubuntu

- We want to experiment a little bit in a clean environment and therefore have a Ubuntu container.
- We launch the default Ubuntu image which will be fetched from the Hub
- `docker run -it ubuntu`
 - Why special options?
 - `-i` gives us an interactive session (attaches `stdin`).
 - `-t` allocates a terminal for us, so we can type in commands.
- Result: `root@cf62f1786c1a:/#`
 - We operate as the *root user* inside the container.
 - The container has a strange default name (its ID)
- We can now do whatever we want!

Using Ubuntu (I)

- Let's install some software.
- Run `apt update` to load newest package lists
- Run `apt install cowsay` to install something ...
- After installation one can run
`/usr/games/cowsay "Docker is a nightmare"`

```
root@2128aaf8db07:/# /usr/games/cowsay "Docker is a nightmare"
  _____
< Docker is a nightmare >
  -----
      \   ^__^
       (oo)\_______
          (__)\       )\/\
              ||----w |
              ||     ||

root@2128aaf8db07:/#
```

Using Ubuntu (II)

- This Ubuntu Image is extremely minimalistic.
- This is good to keep its size down (only 78 MB).
- You need to install everything manually.
 - Good because you are forced to think about your dependencies!
- Remember: a container is a throw-away instance.
- If you run exit, the session will terminate and the container is stopped.
- Once the container is stopped, you cannot access its data anymore.
- You cannot restart a stopped container!

Working with Containers (I)

- Every Container needs a primary process (e.g. the `bash` process in our Ubuntu Container).
- This process needs to be blocking!
- If this process terminates or returns control, the Container terminates.
- In our example, we had a direct keyboard input to the primary process.
- If we want to “detach” the input and return to our host, we can use `Ctrl + P`
- We can “re-attach” again by running `docker attach <Container Name>`.
- Note: This detach/re-attach thing is rarely the way to go. Only use this for the purpose shown here.
- If you want to get a shell in a running Container, do this:
`docker exec -it <Container-Name> /bin/bash`
- Hint: as `<Container-Name>` you may use either the name reported under `docker ps` or a non-ambiguous prefix of the ID!

Working with Containers (II)

- Demonstration of what happens when the primary process terminates:
- Run `docker run ubuntu`.
- What happens? Nothing??
- Not quite:
 - The Ubuntu container is started
 - Main process `/bin/bash` is launched
 - No commands to execute, process terminates
 - Container terminates

Stopping Containers

- When you detach from a running Container or start Containers in the background, `docker ps` still shows you that the Container is up.
- You can stop the Container via the Docker Daemon by running:
`docker stop <Container-Name>` (graceful shutdown)
- Most of the time one rather uses:
`docker kill <Container-Name>`

Cleaning Up

- By default, Docker leaves the remains of stopped Containers on your machine.
- You may want to clean up the files of stopped containers on a regular basis.
- A script that does this:

```
# !/ bin / bash
docker rm -v $( docker ps -a -q -f status = exited )
docker rm -v $( docker ps -a -q -f status = created )
docker rmi $( docker images -f "dangling = true" -q )
```

- `docker rm` removes the stopped containers.
- `docker rmi` removes unreferenced images.
- You can find out about your stopped containers with `docker ps -a`.

Volumes

- Once you stop a Container, all its data is lost!
- There are many use-cases where you want to preserve some data.
 - You run a database server in Docker: $\Rightarrow$ You need to preserve the database.
 - You run a webserver in Docker: $\Rightarrow$ You need to have access to files from the host.
- A *volume* lets you access directories/files from the host inside the Container in a transparent way.
- You tell the Daemon which host directory should be “mounted” where inside the Container.

Volumes Example

- We want to write a program in C on the host, compile it in a container and launch the program on the host again.
- Use case: We may not want to install a compiler on our host.
- Specify volume at container start:
 - `-v /path/on/host:/path/in/container`
 - Example: `-v /tmp/tools/:/opt`
- Specify the working directory at container launch:
 - `-w /path/in/container`

```
$ docker run \  
-v /tmp/tools/:/opt \  
-w /opt \  
gcc \  
gcc prog.c -o myapp
```

Mount the Volume to /opt in the Container
Specify working directory
This is the Image Name
Command to run inside the Container

Writing Dockerfiles

- The Dockerfile is a description/script how an Image is created.
- A Dockerfile can be built/compiled into an Image.
- Every operation in the Dockerfile creates an Image layer, which can be shared among other Images (for reuseability).
- Usual structure and directives:
 - `FROM <OtherImage>` : Specifies a base Image that is used as a foundation. Common Images are debian, ubuntu or alpine.
 - `RUN apt update && apt install -y <package-list>` : Install packages in the new Image (example for Debian/Ubuntu based Images).
 - `COPY /hostpath /dockerpath` : Copy a file from `/hostpath` on the host into `/dockerpath` inside the new Image.
 - `CMD <path-to-program>` : Specify which program should be executed when starting the container (e.g. `/bin/bash`).

Example Dockerfiles

Simplified version of a TexLive Image:

```
FROM ubuntu
WORKDIR /root
ENV TZ=Europe/Berlin
RUN apt-get update && \
    apt-get install -y git unzip curl openssh-client texlive-full
CMD /bin/bash
```

- Name this file Dockerfile (no file extension) and place it in an empty directory.
- We can now build the image with

```
docker build -t myImage /home/user/DirWithDockerfile
```
- This is available on the Docker Hub via the image name `fbeenen/texlive`.
- Writing Dockerfiles can be very complicated – these examples are just a simple basic demonstration.

Networking

- Docker handles the network inside the containers.
- By default (usual case), the containers are attached to a special network (e. g. via the `docker0` interface on the host).
- IP addresses of Containers are managed by Docker.
- Communication to the outside world is done via NAT on the host. Docker can create the required IPTables rules for this.
- You can create new networks managed by docker with the `docker network` command.
- Even more advanced: You can connect a Container to the host network interface, so it shares the IP with the host (do not use this unless required).
- Use `docker network ls` to list available networks.

Thoughts about Security

- *Docker is cool* because you package everything you need in your image.
- *Docker is dangerous* because you package everything you need in your image.
- Example: You build your image based on `ubuntu:latest`. Your app runs for 2 years.
 - You will *never receive security updates* for your runtime libraries because they are bundled in your image!
 - Solution: Re-build all your images on a regular basis (e.g. by a GitLab CI pipeline)!
- Trust issues when using images from the Hub.
 - You trust that this Image does not contain malware/bad things.
 - Also very difficult to guarantee that every sub-Image that you use is clean and built on a regular basis.
 - Solution: Build everything yourself. You can use the Dockerfiles from the Hub, but do not rely on their build!

Oldie but Goldie: <https://www.youtube.com/watch?v=RqhxphyVL88>

Apptainer Overview

- Apptainer is a *container* platform.
- Apptainer is focused on:
 - Effective security model: same user inside a container as outside
 - Integration over isolation: cluster/server system accessible by default
 - SIF container format is easy to share
- Secure alternative to Docker
- Designed for high-performance computing
- Compatible with Docker and OCI images
- Documentation:
<https://apptainer.org/docs/user/latest/>

Apptainer Features

- Provides all benefits of a container platform, e.g.
 - Bundles application together with all required software.
 - Based on single image files that are transferable.
 - Images are independent on library version of the host system
- Does not need a daemon:
 - Easier to handle
 - No security concerns: container runs within the user context
- Bring Your Own Software
 - Install what you want inside the image and then run it.
 - User outside the container is the same user inside.
 - ⇒ Not possible to escalate privileges this way!

Installing Apptainer

- Installed via the distribution's repository or via GitHub <https://apptainer.org/docs/admin/main/installation.html>
- After installation, Apptainer can be started as a normal user
- Command line interface to interact with containers
- From within a container:
 - Redirect I/O, use pipes
 - Pass arguments and shell variables
 - Access files
 - etc.
- Information about interface and individual subcommands
`apptainer help [subcommand]`

Apptainer Bind Mounts

By default, Apptainer activates the following bind mounts:

- User's home directory (`$HOME`)
- Current working directory (`pwd`)
- System mounts:
 - `/dev`
 - `/etc/hosts`
 - `/etc/localtime`
 - `/proc`
 - `/sys`
 - `/tmp`
 - `/var/tmp`
- System configuration files:
 - `/etc/resolv.conf`
 - `/etc/groups`
 - `/etc/passwd`

Running and Converting Docker Images

- Apptainer works with Docker images:
 - 1 Convert “on-the-fly”
 - 2 Permanently convert to Apptainer’s SIF format
- Convert “on-the-fly”

```
apptainer run docker://hello-world
```

Very similar to Docker:

```
:~$ docker run hello-world
```
- Permanently convert an image, e.g. save to hello-world.sif

```
apptainer build hello-world.sif docker://hello-world
```

Disabling and Enabling Bind Mounts

- The option `--no-mount` allows (specific) mounts to be disabled.
- To disable multiple mounts, separate them by commas:

```
apptainer run --no-mount tmp,sys,dev mycontainer.sif
```
- The option `--mount` allows mounting of host directories using a Docker compatible syntax.
- Apptainer provides an easier way of mounting directories via the `--bind` option.
- To bind a directory `/data` on the host to `/mnt` in the container:

```
apptainer exec --bind /data:/mnt mycontainer.sif
```

The directory `/mnt` does not need to exist in the container.

Building Containers from Definition Files

- Apptainer can work with *definition files* that describe how to build a container image. Example:

```
Bootstrap: docker
From: ubuntu:16.04

%post
apt-get -y update
apt-get -y install cowsay lolcat

%environment
export LC_ALL=C
export PATH=/usr/games:$PATH

%runscript
date | cowsay | lolcat
```

- Very similar to Docker definition files
- Creating an image requires root-privileges:

```
sudo apptainer build myimage.sif myimage.def
```

Bring Your Own Software (I)

- Apptainer containers are read-only by default.
- Filesystem overlay mechanism makes immutable images writeable.

- Create an overlay image with 100MB free space

```
apptainer overlay create --size 100 /tmp/overlay.img
```

- Possible to install software permanently without changing the original image

- Create an overlay image with fakeroot option

```
apptainer overlay create --size 100 --fakeroot /tmp/ov
```

fakeroot enables the user to act within the container as root.

Bring Your Own Software (II)

- Using the overlay image, we can apply modifications to the container

```
$ aptainer shell \  
    --overlay /tmp/overlay.img myimage.sif
```

- To install software, root access is required:

```
$ aptainer shell --fakeroot \  
    --overlay /tmp/overlay.img myimage.sif
```

- We can now install software as usual, e.g.:

```
$ apt update  
$ apt install cowsay lolcat  
$ date | /usr/games/cowsay | /usr/games/lolcat
```

Exercise

- Apptainer Exercise