

Containerization *

Jens Lechtenbörger

IT Systems, Summer Term 2024

The topics of virtualization and containerization are addressed in two presentations, of which this is the second one. It focuses on containerization, with Docker as popular implementation.

1 Introduction

Let us start with a quiz.

1.1 Retrieval Practice

Take this quiz.

Agenda

- Part 1
 - [Introduction](#)
 - [History and Variants](#)
 - [Virtualization](#)
- Part 2
 - Containerization
 - Docker
 - Conclusions

Given the basis of virtualization from part 1, we now look at containerization as lightweight variant. In particular, we take first steps with Docker.

2 Containerization

- Containerization = lightweight virtualization

*This PDF document is an inferior version of an OER HTML page; free/libre Org mode source repository.

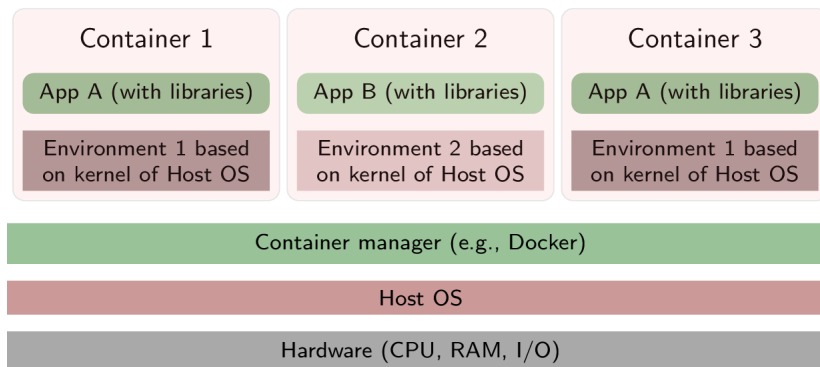


Figure 1: Layering with containerization

- Trade isolation for efficiency (Soltesz et al. 2007)
 - * **Main idea** of containerization: **Share kernel** among containers

Containerization is sometimes considered as lightweight form of virtualization, where the role of virtual machines is replaced with so-called containers, which are created by a container manager. In this presentation, we only consider *Docker* as container manager.

Basically, a container is just a specially restricted process (or set of processes), under control of the container manager.

Being processes, all containers share the same OS kernel, which induces less overhead than the use of a separate OS per VM. Thus, containers are generally more efficient than VMs.

In particular, we can expect that a new container, as process, can be started much faster than a new VM, which essentially boots a new OS.

This gain in efficiency comes at the cost of reduced isolation, as a single OS kernel is shared among all containers.

- Linux mechanisms
 - * **Kernel namespaces**: Limit what is visible inside container
 - * **Control groups (cgroups)**: Limit resource usage
 - * **Own filesystem (chroot)**, copy-on-write, e.g., **UnionFS**:
 - New container without copying all files, localized changes

To isolate different containers from each other, multiple restriction mechanisms are applied, some of which will be shown in class. In case of the Linux kernel, namespaces limit what is visible inside containers. Control groups limit what resources are available inside containers. The filesystem inside a container is largely independent of the host filesystem, may consist of multiple layers, which are overlaid on top of each other, with a copy-on-write mechanism that avoids copying of all files for new containers as long as no write operations occur.

2.1 Terminology

- **Images** specify execution environments

With *Docker*, and other containerization technologies, images specify virtualized execution environments.

- What OS, what components/programs/dependencies?
 - * **Dockerfile** as build recipe for image

- **Reproducibility**

For production use, virtualized execution environments must be deployable in a reproducible fashion to guarantee that software is always executed in its required environment. In particular, the “runs on my machine” syndrome needs to be avoided, where a developer installs all kinds of libraries and dependencies on the development machine but forgets to document necessary pieces. Then, later on, in the production environment, dependencies may be missing or be installed in incompatible versions, potentially leading to subtle bugs or failures.

To prevent such failures, each image is described by a **Dockerfile**, which is a build recipe in a simple text format. It starts from a base image, e.g., an OS variant, and describes what components should be installed.

- E.g. excerpt of **Dockerfile** for TTS that generates audio in this presentation

```
FROM debian:12.1-slim
RUN apt-get update && apt-get install --no-install-recommends \
    curl ffmpeg git-lfs python3-pip python3-venv unzip -y
RUN python3 -m venv /tts
ENV PATH="/tts/bin:$PATH"
RUN pip install wheel
COPY tts/requirements* /tts/
RUN pip install -r /tts/requirements-torch.txt
RUN pip install -r /tts/requirements.txt
...
```

E.g., here you see an excerpt of a **Dockerfile** for the text-to-speech implementation that generates audio for presentations such as this one. (The real **Dockerfile** also contains comments, which are removed here for space reasons. See the full file if you are interested.)

The first line specifies that this image is derived from an image for Debian, which is a GNU/Linux distribution. The second line updates the Debian system and installs additional software, in particular Python. Further lines then set up the Python environment to be used by the text-to-speech implementation.

Thus, essential setup information is contained in a simple text file.

- **Image is template for container**

Images can be seen as templates or blueprints for containers. From a single image, multiple containers can be created on the same computer.

- **Registries publish images**

Images are distributed through registries, from where they can be downloaded. This downloading happens automatically when necessary images are not available locally.

- **Container** is process (set), created from image

As already mentioned, a container is just a specially restricted process (or set of processes), under control of the container manager. The environment for the container is created based on the specification of its image.

2.2 Self-Study Question

- Which conditions for [virtualization as defined in 1974](#) does Docker satisfy?

Take a break to work on the self-study task here.

3 Docker

Make sure to have Docker installed.

3.1 Installation

- Docker Engine ([FLOSS](#), no GUI) is available for different OSs
 - See [here](#) for installation links
- Install on one of your machines, ideally on one that you can bring to (or access in) class
 - Your installation may come with a graphical user interface (GUI), which you do **not** need
 - * Some students perceive the GUI to be confusing
 - * Use command line instead to enter commands shown subsequently (any terminal should work, maybe try `Bash`)

Different Docker variants exist, which share the Docker Engine as basic component. Please install it now if you did not do so as part of a warm-up task already. Subsequently, we only consider Docker commands on the command line.

3.2 Basic Commands

- Start container from image `hello-world`

- `docker run hello-world`

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
[...]
```

Recall the output of the “hello world” example for Docker. Using the vocabulary introduced so far, `docker run` is the command to start a container from the image whose name is given as argument.

In this case, Docker reports that the “hello world” image is not available locally. Thus, it downloads that image from a default registry. Afterwards, it starts a container and runs its code.

If you run the same command again, the container is started immediately, as the image is now available locally.

- List your images and containers

- `docker image ls`

- `docker container ls -all`

- * Help is available, e.g.:

- `docker container --help`

- `docker container ls --help`

Basic Docker commands to list images and containers are shown here. Note that containers have hexadecimal IDs and random names.

Also note that help is available.

- Maybe delete image and container

- `docker rmi -f hello-world`

Images can be removed if they are not needed any more.

3.3 Self-Study: A Web Server

- Run web server `nginx`
 - Web browsers and servers talk `HTTP`
 - `docker run -p 8080:80 nginx`
 - * `-p`: Web server listens on **port** 80 in container; bind to port 8080 on host

As a more interesting example, start the web server `nginx`.

In the context of cloud computing, we will see general concepts of distributed systems, for which web servers are a typical example. Without going into too many details about networking now, network server processes listen for incoming data at so-called ports, which are registered with the OS.

Clients can then connect to those ports. Of course, clients and servers need to talk a common language. Such languages are specified by protocols.

In case of the web, servers and clients “talk” `HTTP` as protocol, for which the default port is 80.

Inside the container, `nginx` asks the environment for a binding with port 80, and a command line option specifies at what port to make the server available on the host. In this case, port 8080 is specified. You can use other port numbers here, but note that ports below 1024 are privileged and require administrator rights.

- Visit `http://localhost:8080`, `nginx` server in container
 - Connect to `nginx` with your browser. As usual, type a URL into your browser. Here, use `localhost` as domain name, which refers to your local computer, and port 8080, separated by colon.
 - You should see a default page of `nginx` in your browser.
- Maybe add option `--name my-nginx`: Assign name to container for subsequent use
 - * E.g., `docker stop/start/logs/rm my-nginx`
 - As a side note, if you add a name when starting a container, you can refer to that name subsequently, e.g., to stop or restart the container or to show log messages.
- Serve own web page, e.g., HTML files
 - Add option `-v` in above `docker run ... (before nginx)`
 - * Mount (make available) directory from host in container
 - E.g.: `-v /host-directory/with/html-files:/usr/share/nginx/html`
 - * `/usr/share/nginx/html` is where `nginx` expects HTML files, in particular `index.html`
 - * Thus, your HTML files replace default ones of `nginx`

By default, the filesystem inside the container is isolated from the host’s filesystem, and it only includes what is embedded into its image. Thus, the web server cannot serve any of your files.

To make a host directory available inside the container, the option `-v` can be used. This option requires two directories, separated by a colon. The first directory is a directory of the host OS, to be made available inside the container under the second directory. In this case, the second directory is the one from which `nginx` serves web resources such as HTML files. Then, `nginx` serves your own files instead of its default page.

If you are not sure what resources to serve, maybe download the ones for our course (several hundred megabytes). Unzip the archive and use the extracted directory as host directory.

Note that subsequent slides contain hints regarding error messages and directory path specifications.

3.3.1 Selected Errors

- **Error** message: name in use already
 - You cannot use the same name multiple times with `docker run --name ...`
 - Instead: `docker start my-nginx`
- **Error** message: port is allocated already
 - You cannot use option `-p` with same port in several `docker run` invocations
 - * Other container still running, stop first
 - `docker ps`: Note ID or name
 - `docker stop <ID-or-name>`
 - `docker run ...`
 - * (Or some other process uses that port. Kill process or choose different port.)

If Docker cannot start a container, it usually displays a helpful error message. Be sure to read it.

Selected issues, and fixes, are shown here.

3.3.2 On Option `-v`

- Say, you start `nginx` with option `-v` but your files do not appear
 - `docker inspect <name-or-id-of-container>`
 - * Check output for `binds`, telling you what is mapped to `/usr/share/nginx/html`
 - May not meet your expectations
 - Are you on Windows?
 - * Try `-v C:\Users\...` with Powershell
 - * Try `-v C:\\Users/...` with Bash
 - * Try `-v /mnt/c/Users/...` with WSL terminal

Depending on your host OS, different path specifications may be necessary with option `-v`. Some variants are shown here.

4 Conclusions

Let us conclude.

4.1 Summary

- **Virtual machines** are **efficient**, **isolated** **duplicates** of real computer
- **Containers** are running processes, defined by **images**
 - Containers on one host share same OS kernel
 - Isolated as processes, with namespaces and cgroups

- Virtual machines and containers
 - can be contrasted in terms of their layering approaches
 - allow deploying software in well-defined environments

Virtual machines are efficient duplicates of a physical computer that provide an isolated environment for running applications. Unlike VMs, which run as separate instances with their own OSs, containers are processes within a single OS.

Each container is defined by its image, which includes all necessary dependencies and libraries required to execute the code.

A key difference between virtual machines and containers lies in their layering approach: While VMs create multiple layers of isolation using different OSs, containers use a shared OS kernel but maintain separation through special kernel features. Both, virtual machines and containers, offer benefits when it comes to deploying software in well-defined environments, providing predictable behavior and reducing compatibility issues.

4.2 Outlook

- Containerization is enabler of **DevOps**
 - DevOps = Combination of Development and Operations (Jabbari et al. 2016; Wiedemann et al. 2019)
 - * Bridge gaps between teams and responsibilities
 - * Aiming for rapid software release cycles with high degree of automation and stability
 - Trend in software engineering
 - * Communication and collaboration, continuous integration (CI) and continuous deployment (CD)
 - * Approach based on Git also called GitOps (Limoncelli 2018)
 - Self-service IT with proposals in Git pull requests
 - Infrastructure as Code (IaC)

The technology of containerization, combined with version control systems like Git, has emerged as enabler of DevOps practices. At its core, DevOps represents a cultural shift towards better communication and collaboration between development and operations teams, aiming to bridge the traditional gap between these two roles. By embracing DevOps principles, organizations can streamline their software delivery process, enabling faster release cycles with higher degrees of automation and stability.

Containerization in combination with version control enables continuous integration and continuous deployment of new versions of software in automated ways. Under the name GitOps, this approach may offer self-service IT capabilities. Additionally, Infrastructure as Code principles allow teams to define infrastructure configurations using declarative configuration files, further simplifying the deployment and maintenance of modern cloud architectures.

Bibliography

- Jabbari, Ramtin, Nauman bin Ali, Kai Petersen, and Binish Tanveer. 2016. “What is DevOps? A Systematic Mapping Study on Definitions and Practices.” In *Proceedings of the Scientific Workshop Proceedings of Xp2016. Xp ’16 Workshops*. <https://doi.org/10.1145/2962695.2962707>.
- Limoncelli, Thomas A. 2018. “Gitops: A Path to More Self-Service It.” *Commun. Acm* 61 (9): 38–42. <https://doi.org/10.1145/3233241>.
- Soltész, Stephen, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. 2007. “Container-Based Operating System Virtualization: A Scal-

able, High-Performance Alternative to Hypervisors.” In *Acm Sigops Operating Systems Review*, 41:275–87. 3. ACM.

Wiedemann, Anna, Nicole Forsgren, Manuel Wiesche, Heiko Gewalt, and Helmut Krcmar. 2019. “Research for Practice: The DevOps Phenomenon.” *Commun. Acm* 62 (8): 44–49. <https://doi.org/10.1145/3331138>.

The bibliography contains references used in this presentation.

License Information

Source files are available on GitLab (check out embedded submodules) under free licenses. Icons of custom controls are by @fontawesome, released under CC BY 4.0.

Except where otherwise noted, the work “Containerization”, © 2019, 2021, 2024 Jens Lechtenbörger, is published under the Creative Commons license CC BY-SA 4.0.

This presentation is distributed as Open Educational Resource under freedom granting license terms.