Docker

Kurzgesagt - In a Nutshell

About Me

- Apprenticeship System Admin/Engineer
- CS Degree at BFH
- Consulting Years
  - Public Cloud Provider, Telco Provider, Medtech
  - DevOps / Automation Engineer, Software Engineer
  - .NET Core, Java Spring Boot and a lot of Tooling
- Securiton
  - Intrusion Alarm System
  - Software Engineer
  - C++, Go and a lot of Tooling

Christian Nydegger
LinkedIn

Intro

Goals of today's lecture

- You can classify Docker
- You know the basic concepts of Docker
- You can apply those concepts
- You know about Docker-Compose

What are Containers?

What are Containers?

A container is a ***standard unit of software*** that ***packages up code and all its dependencies*** so the application runs quickly and reliably from one computing environment to another.

- Wait.. that just sounds like an ordinary software package?
  - Yes but it is so much more
- A Software package can be
  - an executable like MS Word
  - a library like Flask or Numpy
  - a simple python script
- An application often depends on libraries

- Depending on the source, the definition of a software package varies. There are sources stating that a software package is a collection of software products. However, in such cases it is more accurate to speak of a software suite than a software package. An example could be Microsoft Office.
- A software package can be an executable like Firefox, but this is not necessarily always the case. It can also be a library package containing a collection of types and functions.
- The key takeaway of this slide is that containers have something to do with with packaging code and its dependencies.

Sources:
https://www.docker.com/resources/what-container

What are Containers?

OS-level ***virtualization*** is an operating system (***OS***) paradigm in which the ***kernel*** allows the existence of ***multiple isolated*** user space ***instances, called containers***..

- So it is a virtual machine?
  - Yes, in a way :)

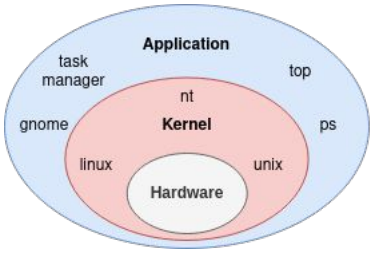
- The term OS-Level virtualization may sound confusing but will be explained over the course of the next few slides.
- The key takeaway of this slide is that containers have something to do with virtualization.

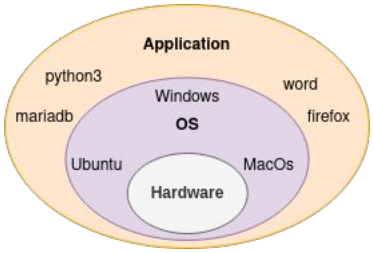
Sources:
https://en.wikipedia.org/wiki/OS-level_virtualization

What are Containers?

What defines an Operating System?

- Kernel abstracts Hardware
- OS is more than a Kernel
  - It also includes various tools like



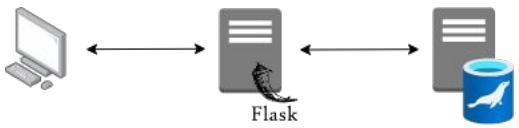


- Bare metal programming, also known as interacting directly with hardware, is difficult and requires a very specific type of programmer with a lot of expertise in both simple programming languages and computer architecture.
- The kernel is a software-based abstraction of the hardware that provides programmers with a rich API. It offers many things like process or memory management. The complexity of a modern kernel like the Linux kernel is unique and very few people actually work at this level (compared to the number of web developers).
- An operating system usually consists of a kernel and additional software that runs "on top" of the kernel. In addition to a kernel, common operating systems such as Windows, Ubuntu or MacOS usually offer a graphical user interface and a rich collection of tools.
- For the purposes of this lecture we will use the following definition of an operating system. It consists of everything you have on your system after a clean installation of the operating system, with no additional packages installed. This may include a graphical user interface, but does not necessarily have to be the case. Linux servers usually only offer a terminal with no graphical user interface.
- This mix of kernel and additional software that defines an operating system is visualized by mixing the colors red (kernel) and blue (application), resulting in purple.

What are Containers?

Project Setup

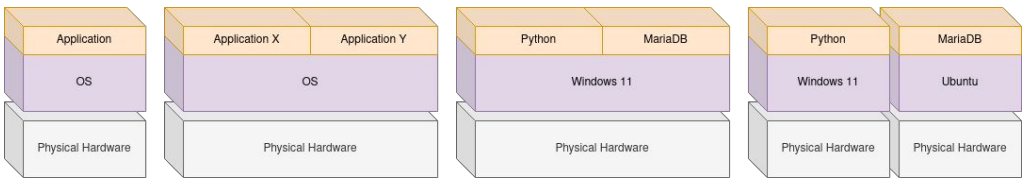
- Simple Web Application Setup
- Python/Flask to implement Rest Service
- SQL DB as Persistence Layer



What are Containers?

Bare Metal Machine

- Application services
  - On same host
    - Little to no isolation on process level
  - On different hosts
    - Isolation by own hardware and os



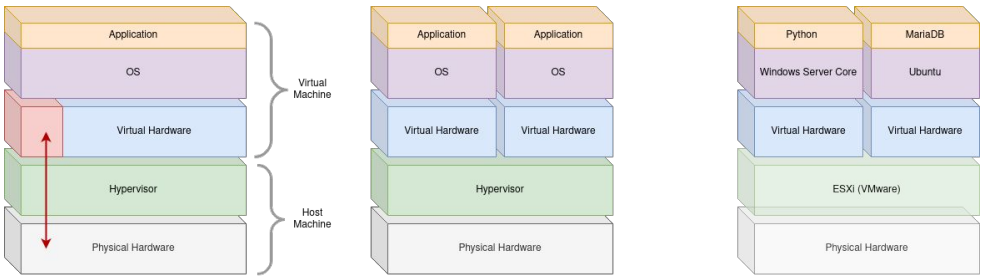
- A setup like it is visualized is most common in consumer electronics like your notebook or mobile phone.
- Most modern software packages like Word, Edge or Firefox are heavily dependent on libraries. If you have access to a Debian based system like Ubuntu, open a terminal and type apt-cache depends firefox and you will see that it has many dependencies.
- Libraries by themselves are usually useless, but very important to avoid code duplication. It would be very inefficient to keep rewriting basic functionalities like a random number generator. When a library is used by a program, the program has it as a dependency. There are different approaches how to deal with dependencies.
- One way is to put a program and all its dependencies together in one big package. A popular exponent of this approach is Snap. The disadvantage is that dependencies may be used by more than one application, each with its own copy.
- Another approach tries to work around this problem by using shared dependencies. This means if two applications depend on the same library, it will only be installed once, but will be used by both. A common exponent of this approach is dpkg (Debian package), which is still the default on most Debian-based systems such as Ubuntu.
- The disadvantage of shared dependencies is that conflicts or incompatibilities can arise. So if an application depends on a specific library version that is not available on the package repository, the application fails to install. Considering that most dependencies have dependencies of their own, this can be a real

- problem. If you're running a stable Debian release (tested and approved by the Debian project), you rarely have the latest version of an application because it has dependencies that aren't available.

What are Containers?

"Traditional" Virtual Machine

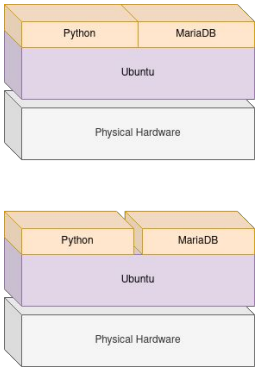
- Isolated Instance with its own Hardware and OS



- Hypervisors are a complicated subject in themselves, but for the purposes of this lecture, it is sufficient to know that they enable virtualization by introducing a virtual hardware layer.
- Simulating/emulating multiple virtual machines generates a fair amount of overhead. To optimize resource efficiency, most modern hypervisors support direct hardware access in certain cases (red arrow on the left most graphic).
- Multiple operating systems can run on the same host/machine, with each instance having its own isolated virtual hardware.
- Technically, the operating system is unaware that it's running on virtual hardware provided by a hypervisor. So from the perspective of the application and the operating system, the hypervisor and the physical hardware layer do not exist. In practice, you often install some tools on the virtual machine that communicate with the hypervisor. However, this is a detail and not so important in the context of this lecture.
- With this form of virtualization it is possible to run different kind of operating systems on the same host which is visible on the left most graphic.
- The MariaDB service and the Python service each run in their own virtual machine and are therefore isolated and cannot interfere with each other.
- Both services can be hosted on individual bare metal machines to achieve isolation, but in practice this is not common as it is simply less flexible.
- Another reason is resource economics. If the workload is not evenly distributed, a machine can have a lot of idle time. This may seem irrelevant, but on a large scale it becomes crucial to run systems in an efficient manner.

What are Containers?

- Linux Process
  - A running program
  - Isolated memory space etc.
- Linux Container
  - A process or group of processes
  - Further isolated by private root-fs, process namespace etc.
  - Enabled by kernel features like cgroups or namespaces
  - OS-Level or Kernel-Level virtualization

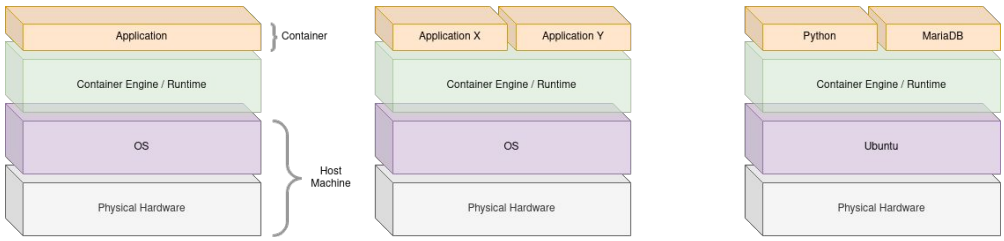


- The effort involved in virtualizing hardware and running multiple operating systems is still too high in many cases. This is where containers come into play. They are lightweight without sacrificing the desired features that a virtual machine offers, namely isolation and flexibility.
- Unlike virtual machines, the isolation is not achieved through virtualization of hardware, but through a handful of features offered by the operating system, more specifically the kernel itself. Hence the name OS-level virtualization. A container is basically an ordinary process which is isolated by different linux kernel features.
- The central feature cgroup was introduced into the Linux kernel in 2008 by Google engineers. Given that Google is constantly running millions of services, resource economics is an important issue for them and it's no surprise that they need such technology to improve the overall efficiency.
- One important thing to point out is that this is a Linux feature only, there are no Windows or Unix based containers! The reason that Windows has support for Docker is a feature called Windows Subsystem for Linux. WSL is basically a virtual Linux machine running on Windows. Same for macOS.
- In theory, a container can also be created by hand without a container engine, but it's not very convenient. And this is where container engines/runtimes come into play.

What are Containers?

Container

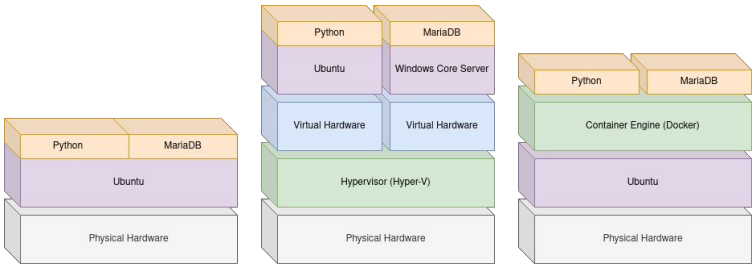
- No overhead of Virtual Hardware or Multiple Operating Systems
  - Isolation is achieved by kernel features not virtualized hardware



What are Containers?

Bare Metal vs. Virtual Machine vs. Container

- More flexibility
- More efficient
- More convenient



Break

If there are any questions, feel free to approach me

Docker Intro

- Set of Tools to work with Containers
- Alternatives
  - Podman
  - LXC
- Why Docker?
  - Well established
  - Big Community
- Terminology
  - Container
  - Image
  - Dockerfile
  - Registry

Docker Intro

Container

- Runtime instance of a Docker Image
- Can be compared to an Object

Docker Intro

Image

Docker images are the basis of containers. An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered filesystems stacked on top of each other. An image ***does not have state and it never changes***.

- Blueprint to instantiate Containers from
- Can be compared to a Class

Sources:
https://docs.docker.com/glossary/

Docker Intro

Dockerfile

A Dockerfile is a text document that contains all the commands you would normally execute manually in order to build a Docker image.

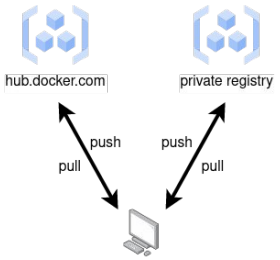
- Instructions for Docker to build Image
- Declares how the Docker Image looks like
- A human readable representation of the Docker Image

Sources:
https://docs.docker.com/glossary/

Docker Intro

Registry

- Hosts Docker Images
  - Can be searched by *docker search*
- Default is hub.docker.com
  - Can be accessed by browser
- Private registry can be setup
  - Available as an Image itself



Docker Intro

Demo

Docker Intro

Process

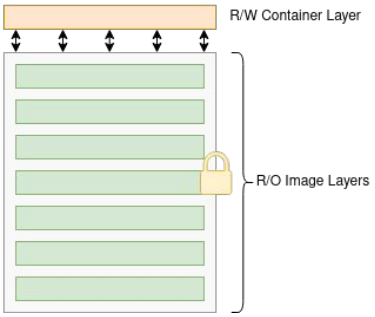
- Write Dockerfile
- Build Image from it
- Instantiate Image to run Container
- Push Image to Registry if desired

Docker Intro

Layer Concept

- Image consists of ReadOnly Layers
- Container ReadWrite Layer represents Container State

```
CMD [ "python", "./main.py" ]
COPY main.py ./
RUN pip install --no-cache-dir -r requirements.txt
COPY requirements.txt ./
WORKDIR /app
EXPOSE 5000
FROM python:3-alpine
```



Docker Intro

What not to do

- Treat a Container like a Virtual Machine
- Upgrade Containers
  - internals
  - Upgrade Dockerfile and rebuild Image instead
- Reuse Containers
  - Run a new container instead
  - If a container is gone, let it rest
- Run multiple Services in on Container
  - Run a container for each service instead

Break

If there are any questions, feel free to approach me

What problem might occur with Docker?

- Docker manages individual Containers
- Dockerfile describes a single Image / Service
- Can become messy for a System that consists of multiple services
    - Starting, Configuring individual Containers
    - Scripts are hard to maintain and not in line with the declarative nature of a Dockerfile

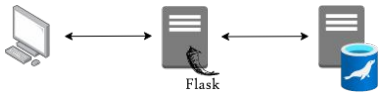
Docker-Compose Intro

Compose is a tool for defining and running multi-container *Docker* applications.

- Compose File
  - Instructions for Compose to configure and run individual Services
- Similar command set as Docker
  - Application level:
    - Up, Down, Build, ..
  - Container level:
    - Start, Stop, Run, ..

Docker-Compose Intro

- CLI
  - Instantiate individual Containers with *docker run*
  - Very inconvenient and error prone
- Script
  - Essentially wrap individual commands in a bash script
  - Technically possible
  - Scripting vs. declaring
- Compose File
  - Declare your multi container application



Docker-Compose Intro

Demo

Your Task

Dockerize a small web application

The goal is to implement a tiny web service similar to the examples discussed during the lecture. It can be a simple ping or something a bit more sophisticated. The only requirement for the service is that the persistence layer is used. The example discussed during the lecture implemented a simple hit count stored in a mariadb.

Other requirements are:

- The rest service and all its dependencies **must** be packed in a Docker Image.
- The persistence layer (mariadb in the example) **must** be run as a container
- The database **must** be sql based
- The state/data **must** be persisted after the db container terminates
- The application **can** be managed with docker-compose (gonna make things easier)
- It is **recommended** to use mariadb and python/flask but it is **not a must**

Deliverables:

- Create one GIT-Repository per group and hand-in at least one solution

Q&A