Praxisgrundlagen der Informatik

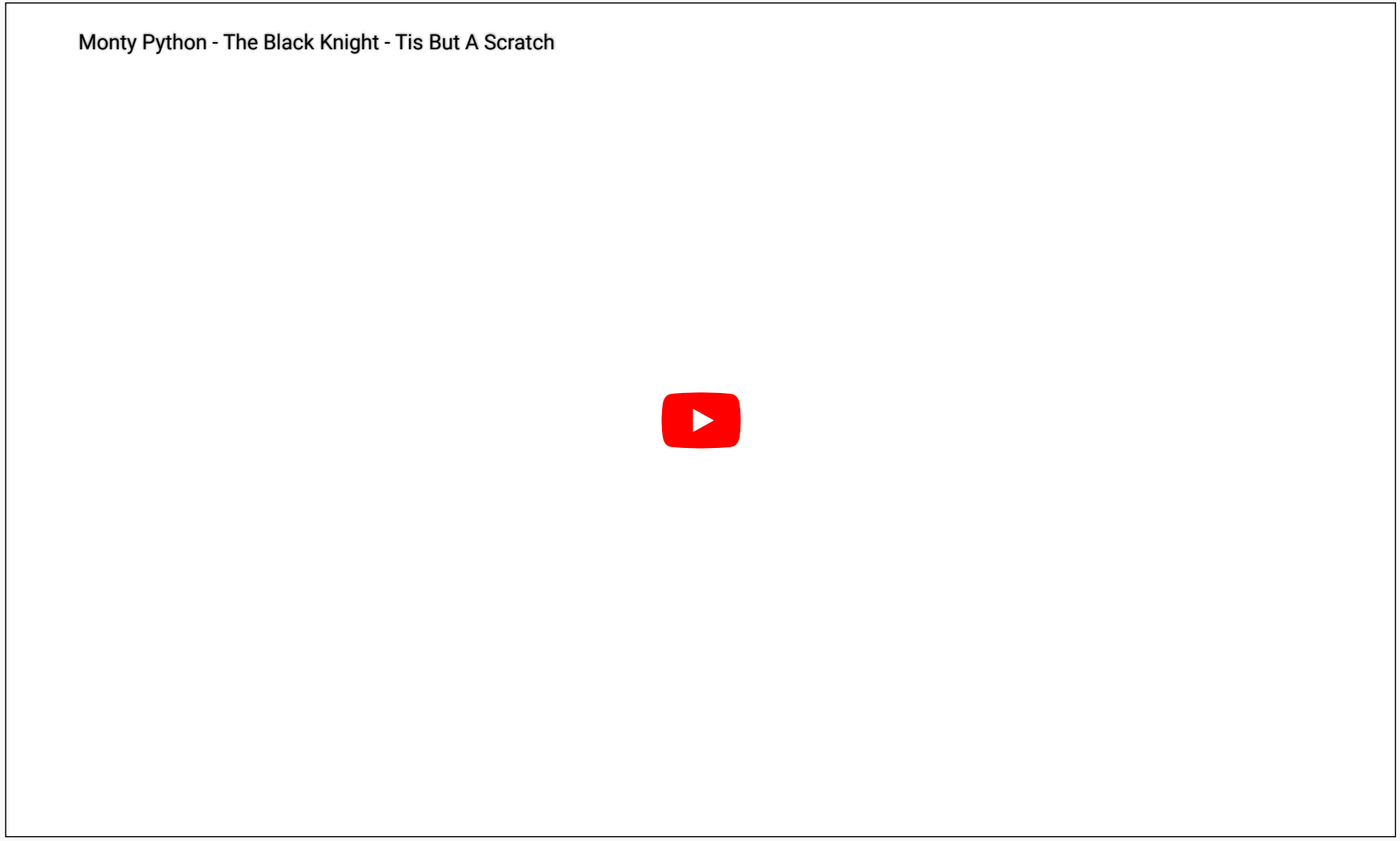
Virtual Environments, Virtual Machines and Containers

Prof. Dr. Eduard Kromer
University of Applied Sciences Landshut

Summer Semester 2024

Virtual Environments

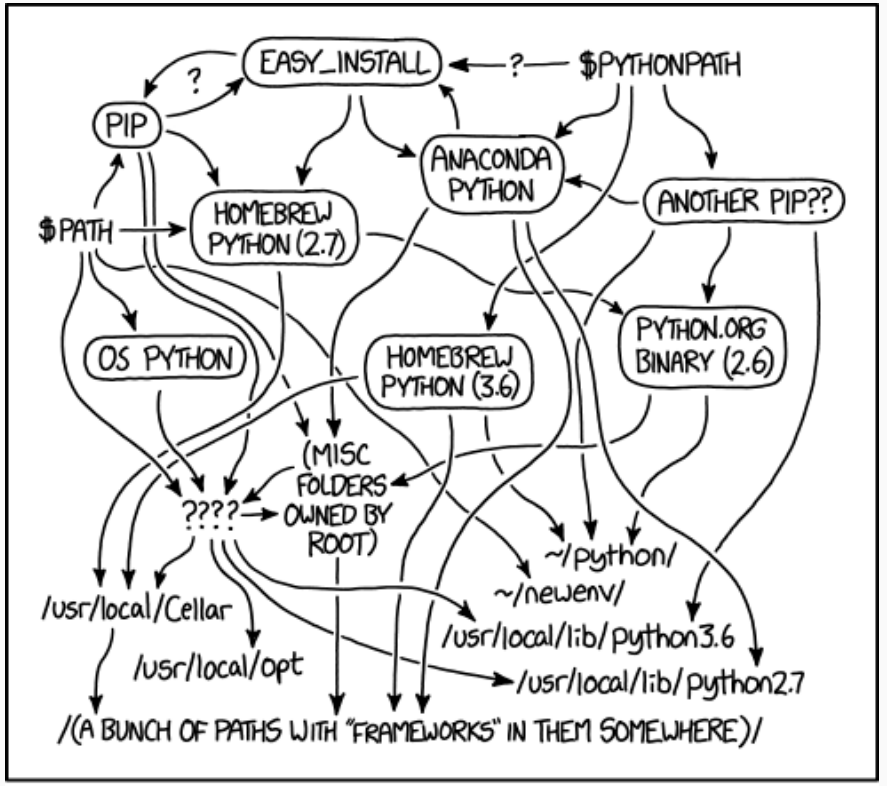
(Monty) Python



Video Source: https://www.youtube.com/embed/ZmInkxbvlCs

Python Ecosystem

How do we manage dependencies?



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Image Source: https://xkcd.com/1987/

How can we use this vast ecosystem effectively?

- we need a CPython distribution:
  - Anaconda / Miniconda
- we need a way to install packages (bundle of software to be installed)
  - conda (mamba)
  - pip
- we need a way to create and manage virtual environments for different projects
  - conda
  - virtualenv and pipenv
  - poetry

What are *virtual environments*?

A virtual environment is

- an environment for Python projects
- a directory with the following components
  - `site_packages` / directory where third-party libraries are installed
  - links (symlinks) to the executables on your system
  - scipts that ensure your code uses the interpreter and site packages in the venv

Virtual Environments

Pros

- reproducible research
- explicit dependencies
  - each project can have its own dependencies (and even different Python versions)
  - there are no limits to the number of environments (they are just directories)
- improved engineering collaboration

Cons

- this is not real isolation
- often does not work across different OSes
- environment setup can be difficult

Managing Virtual Environments with Conda

For this course we will use conda (part of miniconda) to create virtual environments:

```
conda create -n pgi python=3.11
```

- check for available environments: `conda env list`
- activate specific environment (here: `pgi`): `conda activate pgi`
- deactivate environment (return to `base` env): `conda deactivate`
- update all packages in a specific environment (here: `pgi`): `conda update -n pgi --all`
- remove environment: `conda remove -n pgi --all`
- clean up your conda installation / remove unnecessary packages: `conda clean -a -y`

Building Identical Virtual Environments (only for the same OS)

Redirect the output of `conda list --explicit` to a `spec-file.txt`

```
conda list --explicit > spec-file.txt
```

- recreate an identical environment (on same OS) from this file

```
conda create --name pgi --file spec-file.txt
```

- install the listed packages in `spec-file.txt` into an existing environment

```
conda install --name pgi --file spec-file.txt
```

The Virtual Environment for this Course

- activate your environment `pgi`:

```
conda activate pgi
```

- install the following packages from the *default* channel

```
conda install numpy scipy matplotlib pandas cython line_profiler memory_profiler
```

- and `jupyterlab` from the *conda-forge* channel

```
conda install jupyterlab -c conda-forge
```

Exporting the Environment: environment.yml

```
conda env export --from-history > environment.yml
```

```
name: pgi
channels:
  - defaults
dependencies:
  - python=3.11
  - cython
  - line_profiler
  - matplotlib
  - memory_profiler
  - numpy
  - pandas
  - scipy
  - jupyterlab
```

Exporting the Environment: environment.yml (2)

Modify the content of `environment.yml`:

- add the `conda-forge` channel
- specify the version numbers of the libraries we installed earlier

Recreate the environment with

```
conda env create -f environment.yml
```

environment.yml

```yaml
name: pgi
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.11
  - cython=3.0.8
  - line_profiler=4.1.1
  - matplotlib=3.8.3
  - memory_profiler=0.61.0
  - numpy=1.26.4
  - pandas=2.2.0
  - scipy=1.12.0
  - jupyterlab=4.1.1
  - joblib=1.3.2
```

Further Reading on Virtual Environments

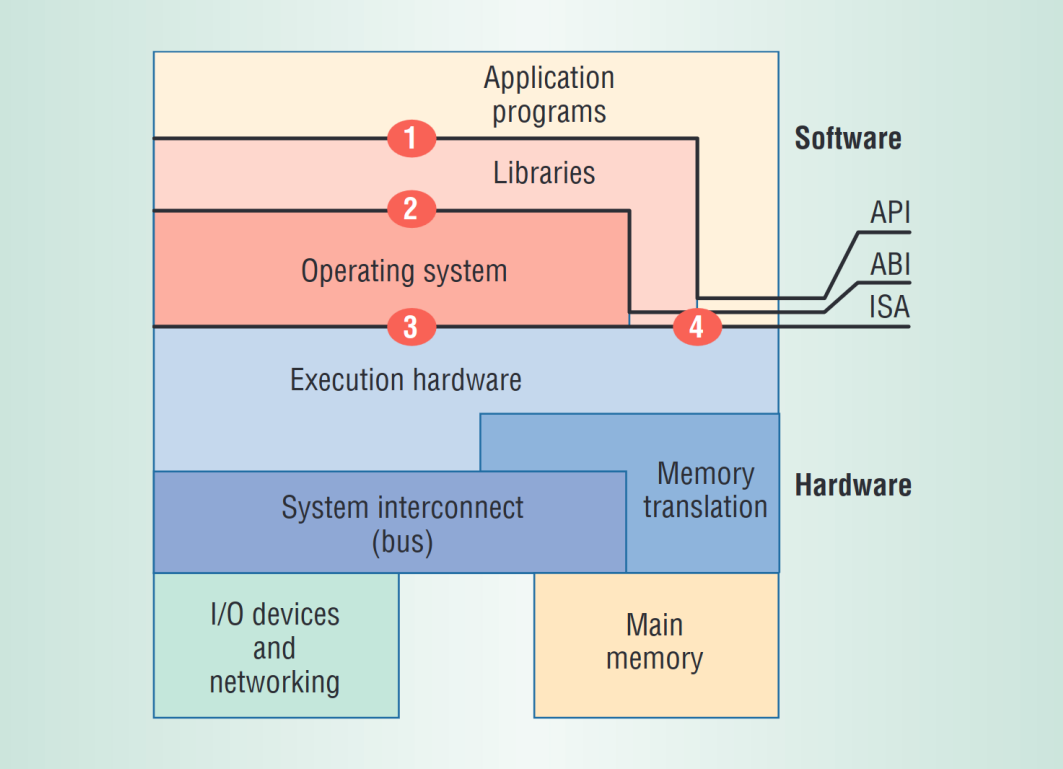
- Getting started with Python environments (using Conda)
- Jake VanderPlas: Conda Myths and Misconceptions

Virtual Machines

Why Virtualize?

- consolidate machines (energy, maintenance, cost, and management savings)
- isolate performance, security, and configuration
- flexibility (rapid provisioning of new services, easy failure / desaster recovery)
- cloud computing (economies of scale)
- ...

Computer System Architecture



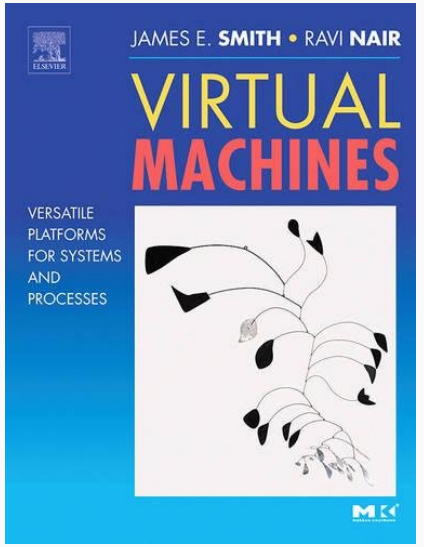
Key implementation layers communicate vertically via the

- instruction set architecture (ISA),
- application binary interface (ABI), and
- application programming interface (API).

J. E. Smith and Ravi Nair, "The architecture of virtual machines," in Computer, vol. 38, no. 5, pp. 32-38, 2005

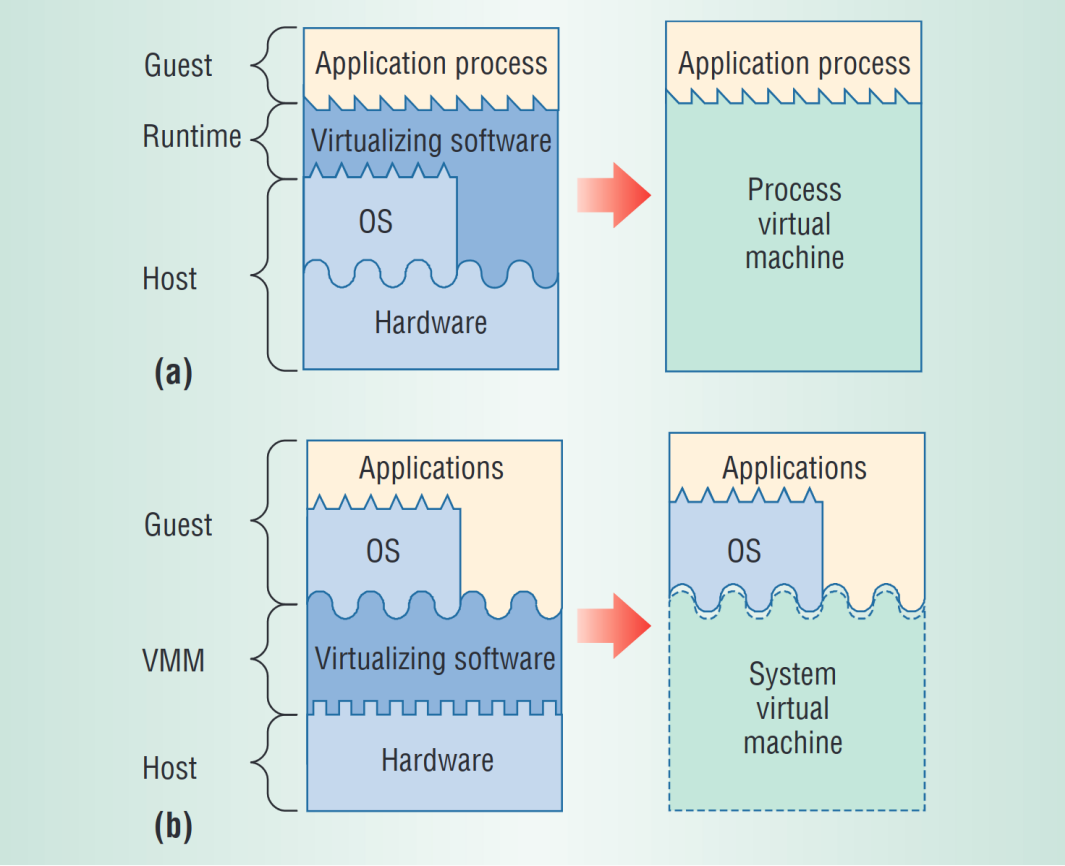
What are Virtual Machines (VMs)?

- Virtual Machines (VMs) eliminate real machine constraints and enable a much higher degree of portability and flexibility.
- A virtual machine is implemented by adding software to an execution platform to give it the appearance of a different platform or for that matter, to give the appearance of multiple platforms.
- A virtual machine may have an operating system, instruction set, or both, that differ from those implemented on the underlying real hardware.



J. E. Smith and Ravi Nair, "Virtual Machines: Versatile Platforms for Systems and Processes", Morgan Kaufmann Publishers, 2005

Types of Virtual Machines - Process VMs

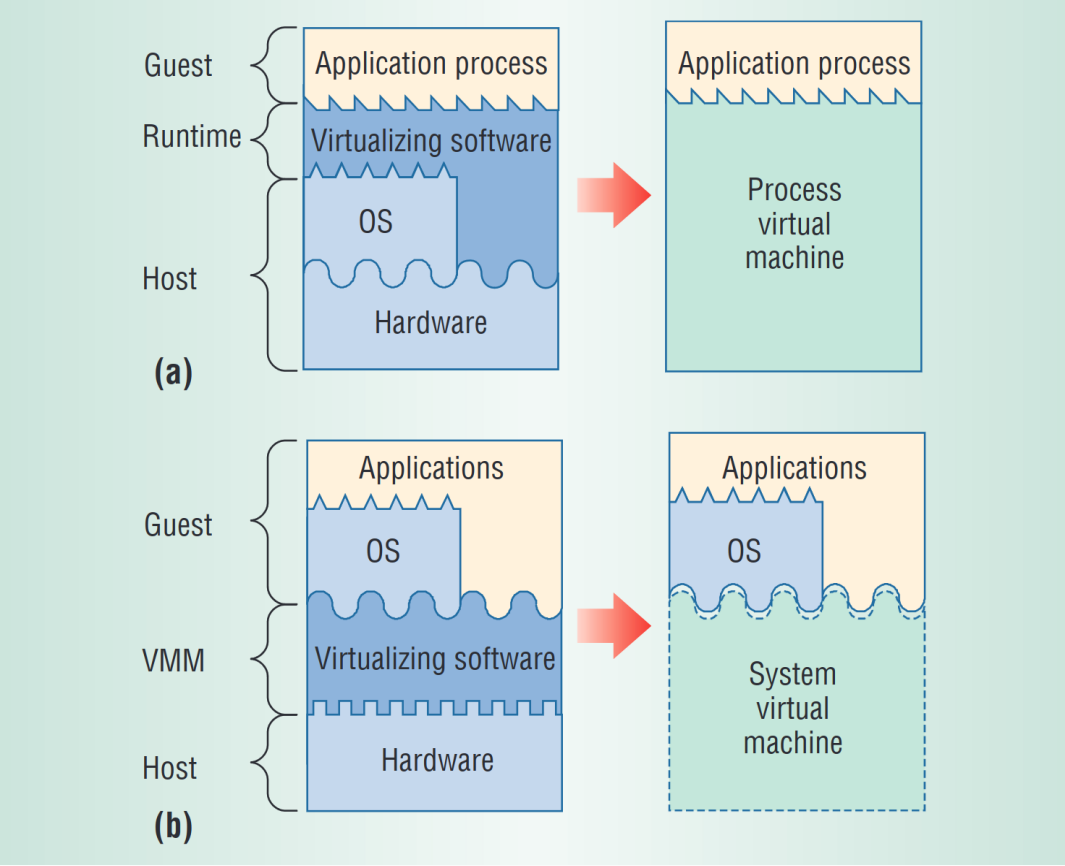


a. Process VM

- virtualizing software translates a set of OS and user-level instructions composing one platform to those of another
- run a single application according to some standard ABI
- example: Java Virtual Machine

J. E. Smith and Ravi Nair, "The architecture of virtual machines," in Computer, vol. 38, no. 5, pp. 32-38, 2005

Types of Virtual Machines - System VMs



b. System VM:

- virtualizing software translates the ISA used by one hardware platform to that of another
- provide a complete system environment at binary *ISA level*
- Single computer runs multiple VMs, and can support a multiple, different OSes
- example: VMware ESX Server, Xen

J. E. Smith and Ravi Nair, "The architecture of virtual machines," in Computer, vol. 38, no. 5, pp. 32-38, 2005

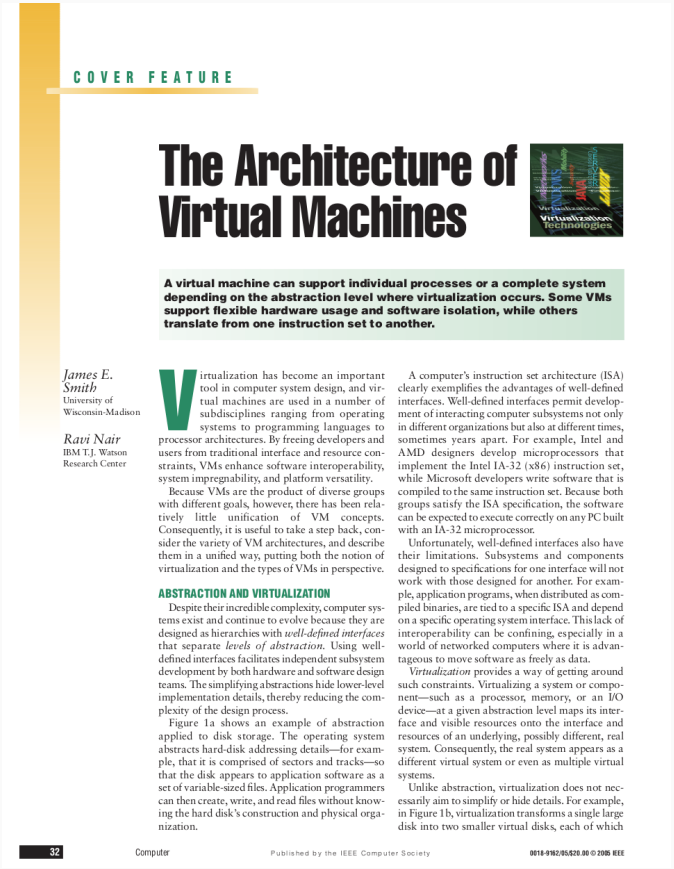
A Hypervisor or Virtual Machine Monitor (VMM)

A *virtual machine monitor* is a control program that can control the execution of a guest program satisfying three requirements:

- The efficiency property: All safe guest instructions are executed by the hardware directly.
- The resource control property: The guest cannot affect control program resources.
- The equivalence property: Guests cannot distinguish whether they are running directly on the hardware or atop the control program.

Homework: Reading Material on Virtual Machines

Read:



COVER FEATURE

The Architecture of Virtual Machines

A virtual machine can support individual processes or a complete system depending on the abstraction level where virtualization occurs. Some VMs support flexible hardware usage and software isolation, while others translate from one instruction set to another.

James E. Smith
University of Wisconsin-Madison

Ravi Nair
IBM T.J. Watson Research Center

Virtualization has become an important tool in computer system design, and virtual machines are used in a number of subdisciplines ranging from operating systems to programming languages to processor architectures. By freeing developers and users from traditional interface and resource constraints, VMs enhance software interoperability, system impregnability, and platform versatility.

Because VMs are the product of diverse groups with different goals, however, there has been relatively little unification of VM concepts. Consequently, it is useful to take a step back, consider the variety of VM architectures, and describe them in a unified way, putting both the notion of virtualization and the types of VMs in perspective.

ABSTRACTION AND VIRTUALIZATION

Despite their incredible complexity, computer systems exist and continue to evolve because they are designed as hierarchies with *well-defined interfaces* that separate *levels of abstraction*. Using well-defined interfaces facilitates independent subsystem development by both hardware and software design teams. The simplifying abstractions hide lower-level implementation details, thereby reducing the complexity of the design process.

Figure 1a shows an example of abstraction applied to disk storage. The operating system abstracts hard-disk addressing details—for example, that it is comprised of sectors and tracks—so that the disk appears to application software as a set of variable-sized files. Application programmers can then create, write, and read files without knowing the hard disk's construction and physical organization.

A computer's instruction set architecture (ISA) clearly exemplifies the advantages of well-defined interfaces. Well-defined interfaces permit development of interacting computer subsystems not only in different organizations but also at different times, sometimes years apart. For example, Intel and AMD designers develop microprocessors that implement the Intel IA-32 (x86) instruction set, while Microsoft developers write software that is compiled to the same instruction set. Because both groups satisfy the ISA specification, the software can be expected to execute correctly on any PC built with an IA-32 microprocessor.

Unfortunately, well-defined interfaces also have their limitations. Subsystems and components designed to specifications for one interface will not work with those designed for another. For example, application programs, when distributed as compiled binaries, are tied to a specific ISA and depend on a specific operating system interface. This lack of interoperability can be confining, especially in a world of networked computers where it is advantageous to move software as freely as data.

Virtualization provides a way of getting around such constraints. Virtualizing a system or component—such as a processor, memory, or an I/O device—at a given abstraction level maps its interface and visible resources onto the interface and resources of an underlying, possibly different, real system. Consequently, the real system appears as a different virtual system or even as multiple virtual systems.

Unlike abstraction, virtualization does not necessarily aim to simplify or hide details. For example, in Figure 1b, virtualization transforms a single large disk into two smaller virtual disks, each of which

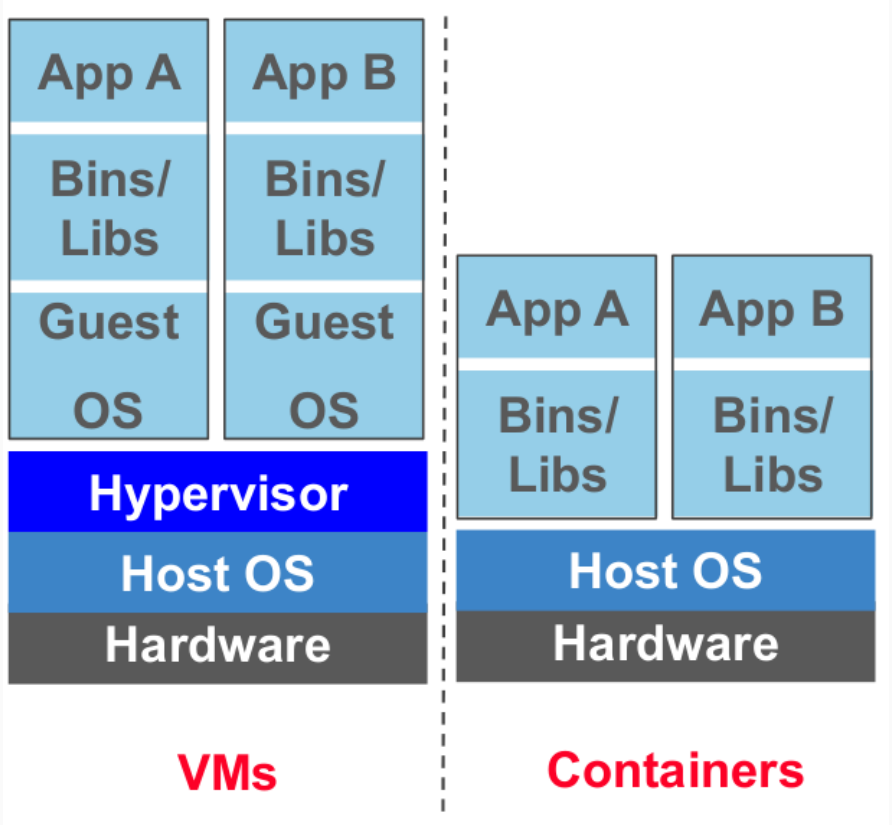
32 Computer Published by the IEEE Computer Society 0018-9162/05/$20.00 © 2005 IEEE

J. E. Smith and Ravi Nair, "The architecture of virtual machines," in Computer, vol. 38, no. 5, pp. 32-38, 2005

Containers

Containers are not Virtualization

- virtual machines provide virtual hardware
- they take a long time to create and require *significant resource overhead*
- they run a whole operating system in addition to the software you want to use



Docker Containers (1)

- Docker containers don't use any hardware virtualization
  - programs inside Docker containers *interface directly with the host's Linux kernel*
  - most Docker containers take less than a second to launch
- Docker helps you use the container technology already built into your OS
- you can run Docker in a virtual machine (if VM uses a modern Linux kernel)

Image Source: https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html

Docker Containers (2)

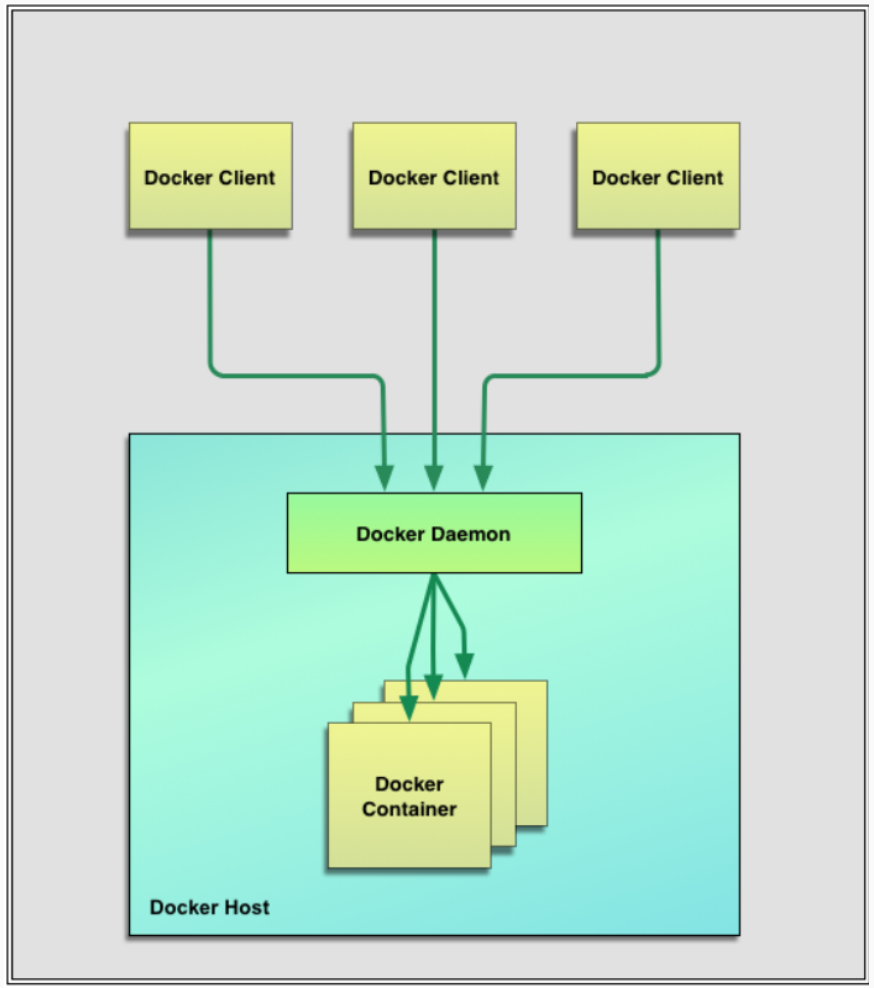
- Docker is designed to ensure that the environment in which developers write code matches the deployment environment for your applications
- it aims to make your application portable, easy to build and to collaborate on
- the microservices architecture encouraged by Docker makes distributing, scaling and debugging your applications easier

What is Docker?

Docker is

- an open source project for building, shipping and running programs
- an open-source engine that automates the deployment of applications into containers
- Docker adds an application deployment engine on top of a virtualized container execution environment

Docker Community Edition Components



- the Docker client and server (Docker Engine)
- Docker images
  - the “source code” for your containers
  - can be shared, stored and updated
- registries
  - Docker stores the images you build in private and public registries; Docker Hub
- Docker containers

Docker Architecture

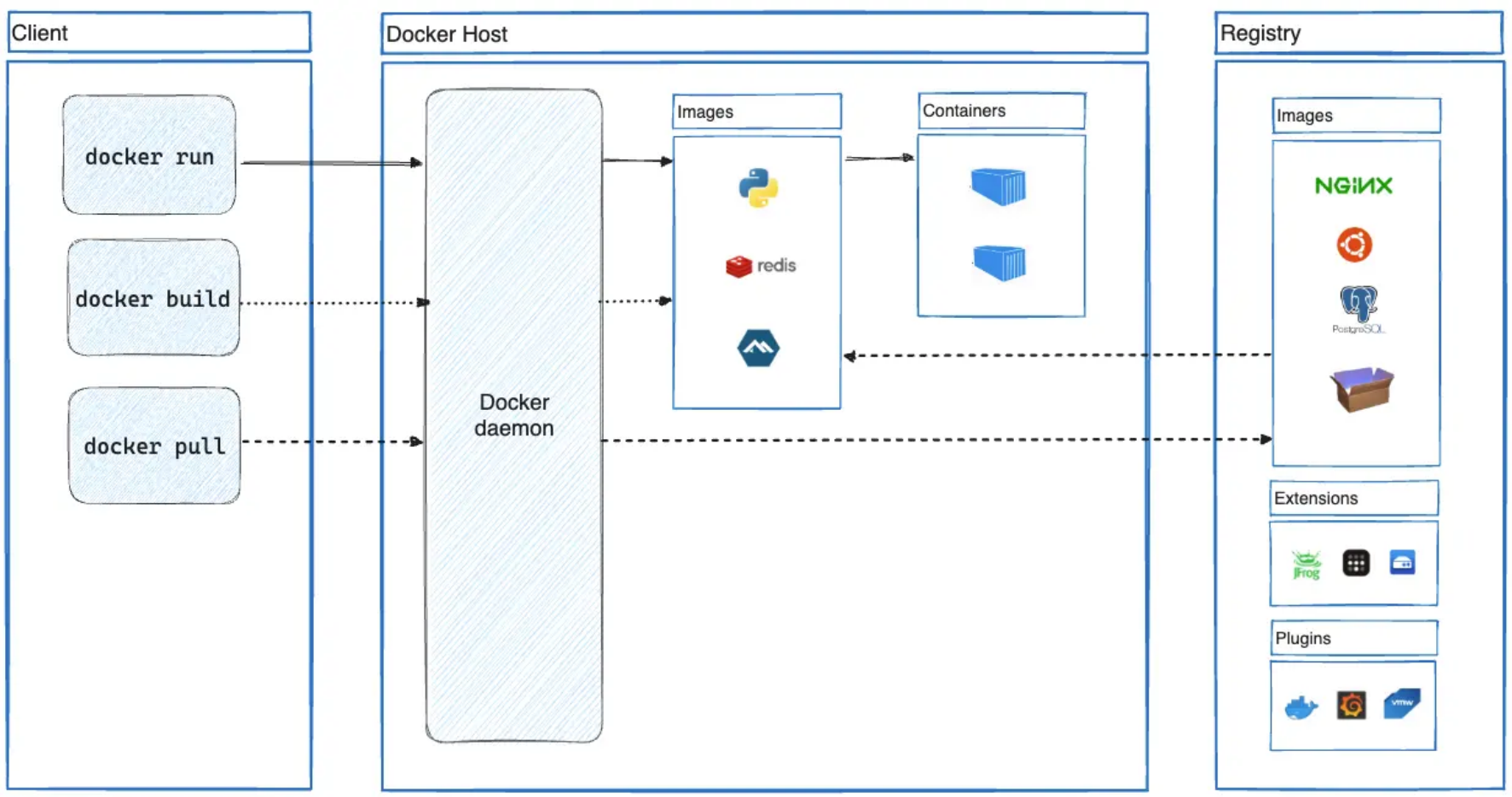


Image Source: https://docs.docker.com/get-started/overview/

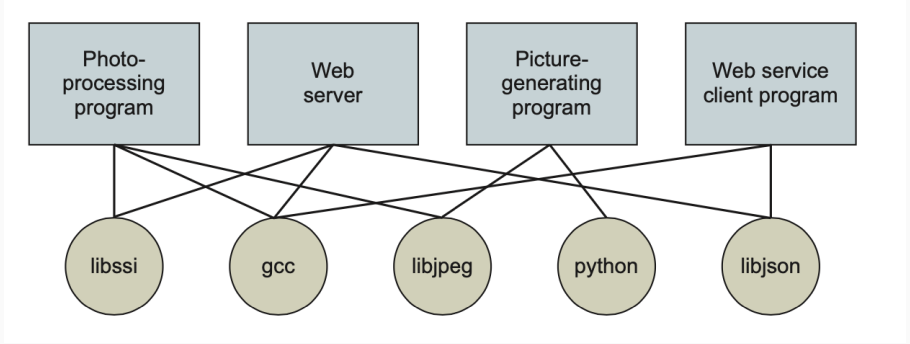
Docker Containers

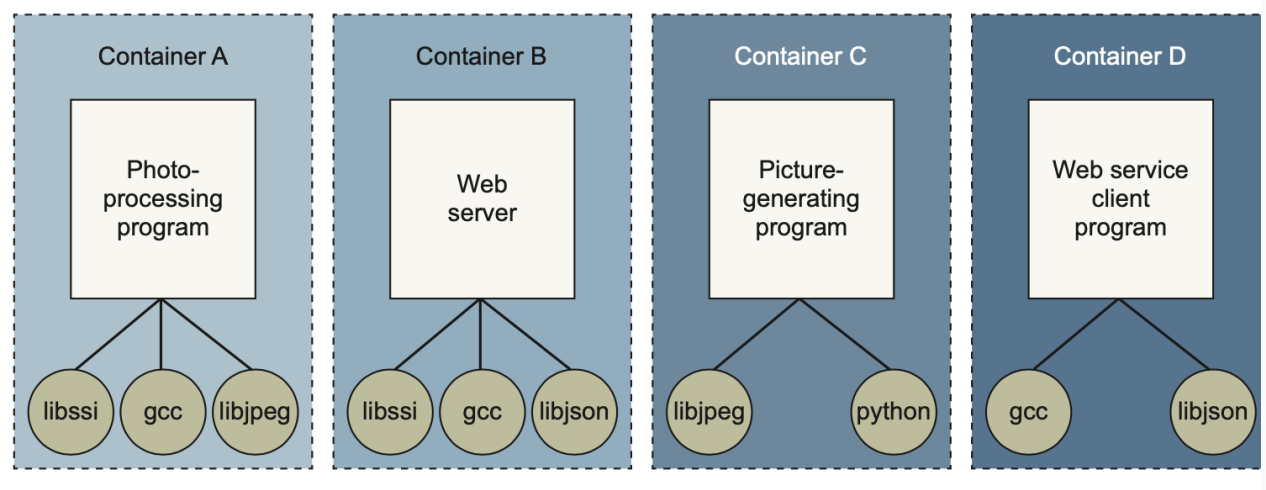
A Docker container is:

- an image format
- a set of standard operations
- an execution environment

- think of a Docker container as a physical shipping container
- a box where you store and run an application and all of its dependencies
- the component that fills the shipping container role is called an image

What problems does Docker solve?





Where and when to use Docker?

- Docker can run almost anywhere — but you should not use it anywhere
- currently, Docker can run only applications that can run on a Linux operating system, or Windows applications on Windows server
- focusing on Linux, you can run almost any application inside a Docker container
- containers *won't help much with the security of programs* that have to run with full access to the machine — containers are not a total solution for security issues

Installing Docker on Ubuntu: Set up Docker's apt repository

Follow the installation instructions here:

1. Set up Docker's apt repository:

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
"deb [arch=$(dpkg --print-architecture) \
 signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu \
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

Installing Docker on Ubuntu: Install the Docker packages

2. Install the (latest) Docker packages:

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

3. Verify that the Docker Engine installation is successful by running the `hello-world` image.

```
sudo docker run hello-world
```

Docker Post-Installation Steps

Manage Docker as a non-root user:

1. Create the `docker` group.

```
sudo groupadd docker
```

2. Add your user to the docker group.

```
sudo usermod -aG docker $USER
```

3. Activate the changes to groups:

```
newgrp docker
```

4. Verify that you can run `docker` commands without `sudo`.

```
docker run hello-world
```

Start Docker on Boot with systemd

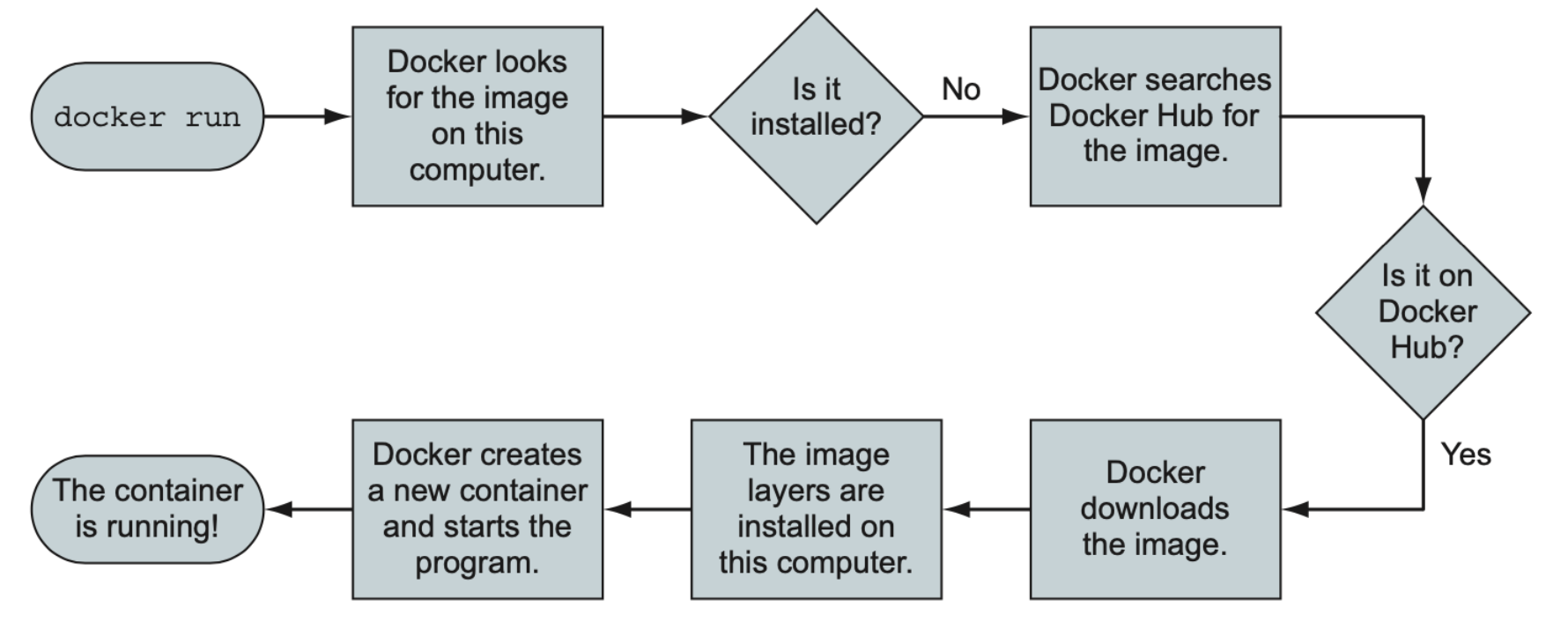
To automatically start Docker and containerd on boot for other Linux distributions using `systemd`, run the following commands:

```
sudo systemctl enable docker.service
sudo systemctl enable containerd.service
```

To stop this behavior, use `disable` instead.

```
sudo systemctl disable docker.service
sudo systemctl disable containerd.service
```

Running a Docker Container



Type `docker help` to find a full list of available commands.

Running a Docker Container

```
docker run -i -t ubuntu /bin/bash
```

- `-i` interactive mode (keep STDIN open even if not attached)
- `-t` allocate a pseudo-tty
- `ubuntu`: this is the name of the (base) image provided on the Docker Hub registry (as an alternative you can choose alpine (only 5MB in size))
- `/bin/bash`: the command to run in our new container — launch a Bash shell

Running a Docker Container

- for a full list of available docker run flags type `docker help run`
- logged in into the container, type: `hostname` or `hostname -I`
- you can now install packages in this container by running

```
apt-get update; apt-get install <packagename>
```

- type exit to stop the container and return to your command prompt

Working with Containers: Show a List of Current Containers

```
docker ps -a
```

shows you a list of current containers.

- you should see a ID, the image used to create it, the command it last ran and much more
- the container has an automatically generated name (you can specify a name for the container with the `--name` flag) — names are unique

Working with Containers: Create, Start and Run

- docker create: creates a fresh new container from a docker image. However, it doesn't run it immediately.
- docker start: will start any stopped container. If you used docker create command to create a container, you can start it with this command.
- docker run: is a combination of create and start as it creates a new container and starts it immediately. In fact, the docker run command can even pull an image from Docker Hub if it doesn't find the mentioned image on your system.

Daemonized Containers

- daemonized containers are ideal for running applications and services
  - with daemonized containers we create longer-running containers
  - they don't have the interactive session we used before
- to start a daemonized / detached container you use `docker run` with the `-d` flag
- by design, containers started in detached mode exit when the root process used to run the container exits; (see documentation for more information)

Daemonized Containers: nginx example

```
docker run -d nginx
```

```
0246aa4d1448a401cabd2ce8f242192b6e7af721527e48a810463366c7ff54f1
```

```
docker ps
```

```
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS         PORTS     NAMES
0246aa4d1448   nginx     "/docker-entrypoint. …"   2 seconds ago    Up 1 second    80/tcp    pedantic_liskov
```

```
docker logs -n 5 0246aa4d1448
```

```
2023/11/06 15:58:23 [notice] 1#1: start worker process 33
2023/11/06 15:58:23 [notice] 1#1: start worker process 34
2023/11/06 15:58:23 [notice] 1#1: start worker process 35
2023/11/06 15:58:23 [notice] 1#1: start worker process 36
2023/11/06 15:58:23 [notice] 1#1: start worker process 37
```

```
docker attach 0246aa4d1448
```

```
^C
2023/11/06 15:58:40 [notice] 1#1: signal 2 (SIGINT) received, exiting
...
```

Container Networking

Containers have networking enabled by default:

- they can make outgoing connections
- you can create a custom network and attach the containers to the network.

```
docker network create my-net
docker run -d --name web --network my-net nginx:alpine
docker run --rm -it --network my-net busybox
/# ping web
```

```
PING web (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.326 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.257 ms
64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.281 ms
^C
--- web ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.257/0.288/0.326 ms
```

Filesystem Mounts

- by default, the data in a container is stored in an ephemeral, writable container layer
- removing the container also removes its data
- if you want to use persistent data with containers, you can use filesystem mounts:
  - volume mounts (for persistently storing data for containers, and for sharing data between containers)
  - bind mounts (for sharing data between a container and the host)

For in-depth discussion, see storage section of the documentation.

Filesystem Mounts: Volume Mounts

```
docker run --mount source=<VOLUME_NAME>,target=[PATH] [IMAGE] [COMMAND...]
```

- the value for the `source` parameter is the name of the volume
- the value of `target` is the mount location of the volume inside the container
  - the `target` must always be an absolute path, such as `/src/docs`

Once you've created the volume, any data you write to the volume is persisted, *even if you stop or remove the container*.

Filesystem Mounts: Bind Mounts

```
docker run -it --mount type=bind,source=[PATH],target=[PATH] busybox
```

- the `source` path is a the location on the host that you want to bind mount into the container
- the `target` path is the mount destination inside the container
- bind mounts are *read-write by default*, meaning that you can both read and write files to and from the mounted location from the container. Changes that you make, such as adding or editing files, are reflected on the host filesystem

Inspecting and Monitoring Containers

- to inspect the processes running inside the container use

```
docker top <ID/name of container>
```

- in addition, you can use the `docker stats` command; it will show you
  - the CPU, memory, network and storage I/O performance and metrics

- to run a process inside an already running container use `docker exec`

```
docker exec -d <ID/name of container> touch /etc/test_config_file
```

Stopping and Deleting Containers

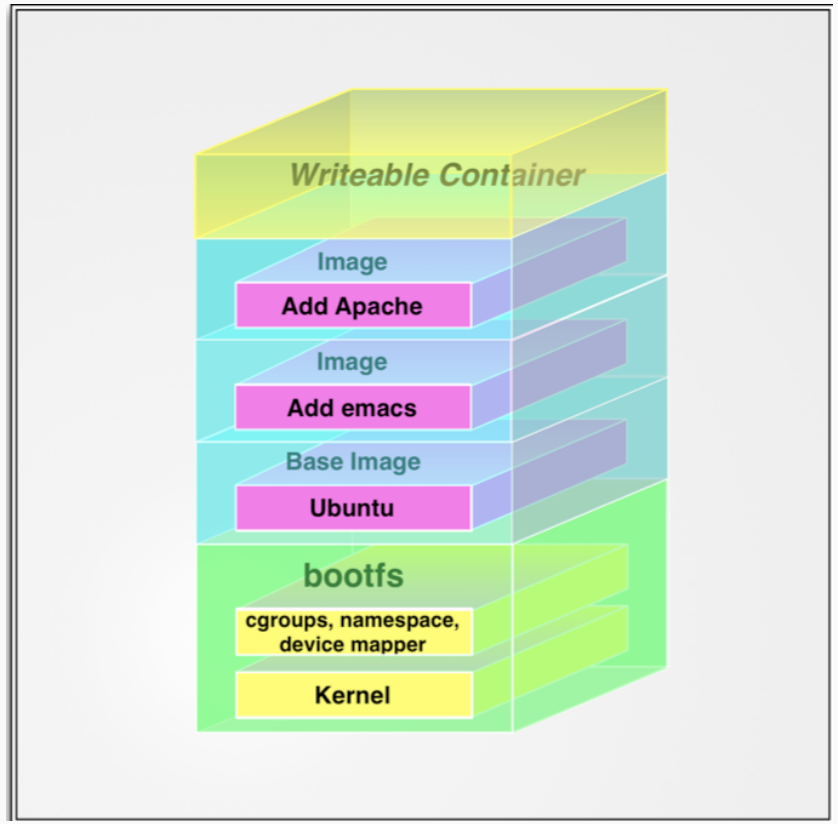
- to stop our detached container we use `docker stop`

```
docker stop <ID/name of container>
```

- if you are finished with a container, you can delete it with `docker rm`
  - if the container is still running, you need to use the `-f` flag

```
docker rm <ID/name of container>
```

Docker Images



- a docker image is made up of filesystems layered over each other
  - bootfs — boot filesystem
  - rootfs — can be one or more OSs (in *read-only mode*)
- when a container is launched from an image, Docker mounts a read-write filesystem on top
  - here processes we want our container to run will execute

Working with Images

- use `docker images` to list all available images on our Docker host
- you should see the `ubuntu` image we downloaded earlier — images live in repositories and repositories on registries (the default one is Docker Hub)
  - the ubuntu repository contains images for Ubuntu 18.04, 20.04, 22.04 etc.
  - you can download images by using
    - `docker pull ubuntu:20.04` or
    - `docker pull ubuntu:latest`

Working with Images: Search for Images

You can search for images with `docker search`

```
docker search ubuntu
```

```
NAME                      DESCRIPTION                                     STARS     OFFICIAL
ubuntu                    Ubuntu is a Debian-based Linux operating sys…   16874     [OK]
websphere-liberty         WebSphere Liberty multi-architecture images …   298       [OK]
open-liberty              Open Liberty multi-architecture images based…   64        [OK]
neurodebian               NeuroDebian provides neuroscience research s…   106       [OK]
ubuntu-debootstrap        DEPRECATED; use "ubuntu" instead                52        [OK]
ubuntu-upstart            DEPRECATED, as is Upstart (find other proces…   115       [OK]
ubuntu/nginx              Nginx, a high-performance reverse proxy & we…   112
ubuntu/squid              Squid is a caching proxy for the Web. Long-t…   83
ubuntu/cortex             Cortex provides storage for Prometheus. Long…   4
ubuntu/prometheus         Prometheus is a systems and service monitori…   56
...
```

Building our own images: Dockerfile

What is a Dockerfile?

- Docker builds images automatically by reading the instructions from a `Dockerfile` which is a text file that contains all commands, in order, needed to build a given image
- the `Dockerfile` uses a basic DSL (Domain Specific Language) with instructions for building Docker images
- for documentation on Dockerfiles and all available commands read the Dockerfile reference

Dockerfile: Example

- you can generate a file with name Dockerfile in some directory
- this directory is called build context (it is assumed that the Dockerfile is located here)
- your Dockerfile can be as simple as:

```
FROM busybox
COPY /hello /
RUN cat /hello
```

Then run

```
mkdir myproject && cd myproject
echo "hello" > hello
docker build -t helloapp:v1 .
```

Dockerfile: Commands

Specific commands you can use in a Dockerfile are e.g.:

- `FROM ubuntu:18:04`: *creates a layer from the ubuntu:18.04 Docker image*
- `COPY . /app`: *adds files from your Docker client's current directory*
- `RUN make /app`: *builds your application with* `make`
- `CMD python /app/app.py`: *specifies what command to run when you launch the built image*

Dockerfile Best Practices

For Dockerfile best practices read:

- https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
- https://www.docker.com/blog/intro-guide-to-dockerfile-best-practices/

Docker Compose

- Docker Compose is an open source Docker orchestration tool for defining and running multi-container applications
- it is written in Python and licensed with the Apache 2.0 license
- Docker Compose relies on a YAML configuration file, usually named `compose.yaml`
  - YAML is used to describe structured documents, which are made up of structures, lists, maps, and scalar values
- the `compose.yaml` file follows the rules provided by the Compose Specification

Docker Compose: Interaction and Installation

- you then interact with your Compose application through the Compose CLI
  - `docker compose up`: *start the application*
  - `docker compose down`: *stop and remove the containers*

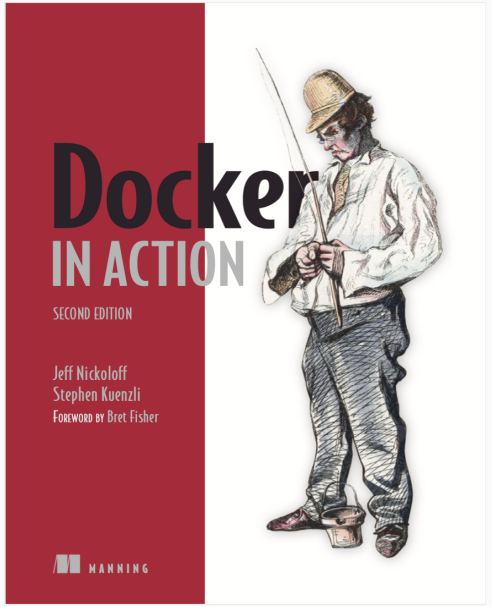
Installation:

```
sudo apt-get install docker-compose-plugin
```

Docker Compose - The Compose File: Example

- Compose files describe every first-class Docker resource type: services, volumes, networks, secrets, and configs
- consider a collection of 3 services: a *PostgreSQL database*, a *MariaDB database*, and a *web administrative interface* for managing those databases

```yaml
services:
    postgres:
        image: dockerinaction/postgres:11-alpine
        environment:
            POSTGRES_PASSWORD: example
    mariadb:
        image: dockerinaction/mariadb:10-bionic
        environment:
            MYSQL_ROOT_PASSWORD: example
    adminer:
        image: dockerinaction/adminer:4
        ports:
            - 8080:8080
```



Docker Compose: An Illustrative Example (1)

Step 1: Define the application dependencies

1. Create a directory `composetest` for the project.
2. Create a file called `app.py` in your project directory and paste the following code in:

```
import time
import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)

def get_hit_count():
    retries = 5
    while True:
        try: return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0: raise exc
            retries -= 1
            time.sleep(0.5)

@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {} times.\n'.format(count)
```

Docker Compose: An Illustrative Example (2)

3. Create another file called `requirements.txt` in your project directory and paste the following code in:

requirements.txt

```
Flask
redis
```

Docker Compose: An Illustrative Example (3)

Step 2: Create a Dockerfile and paste the following code in:

```
FROM python:3.10-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run"]
```

Docker Compose: An Illustrative Example (4)

Step 3: Define services in a Compose file. Create a `compose.yaml` file in your project directory and paste the following:

```yaml
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

- the `web` service uses an image that's built from the Dockerfile in the current directory
- it binds the container and the host machine to the exposed port, `8000`
- the default port for the Flask web server is `5000`

Docker Compose: An Illustrative Example (5)

Step 4: Build an run your app with Compose.

1. Start up your application by running

```
docker compose up
```

```
Creating network "composetest_default" with the default driver
Creating composetest_web_1 ...
Creating composetest_redis_1 ...
Creating composetest_web_1
Creating composetest_redis_1 ... done
Attaching to composetest_web_1, composetest_redis_1
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
redis_1  | 1:C 17 Aug 22:11:10.480 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
redis_1  | 1:C 17 Aug 22:11:10.480 # Redis version=4.0.1, bits=64, commit=00000000, modified=0, pid=1, just started
...
```

Docker Compose: An Illustrative Example (6)

2. Enter http://localhost:8000/ in a browser to see the application running. You should see a message in your browser saying:

```
Hello World! I have been seen 1 times.
```

3. Refresh the page. The number should increment.
4. Switch to another terminal window, and type `docker image ls` to list local images. (`web` and `redis` should show up)
5. Stop the application, either by running `docker compose down` from within your project directory in the second terminal, or by hitting CTRL+C in the original terminal where you started the app.

Docker Compose: An Illustrative Example (7)

Step 5: Edit the Compose file to add a bind mount.

```yaml
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_DEBUG: "true"
  redis:
    image: "redis:alpine"
```

- the new `volumes` key mounts the project directory (current directory) on the host to `/code` inside the container
- `FLASK_DEBUG` environment variable: tells `flask run` to run in development mode and reload the code on change

Docker Compose: An Illustrative Example (8)

Step 6: Re-build and run the app with the updated Compose file with `docker compose up`
Step 7: Update the application code and see changes instantly.

- Change the greeting in `app.py` and save it. For example, change the `Hello World!` message to `Hello from Docker!`
- Refresh the app in your browser. The greeting should be updated, and the counter should still be incrementing.

Your turn: Try other sample apps with Compose at https://github.com/docker/awesome-compose.

GPU-support in Compose

GPUs are referenced in a `compose.yml` file using the `devices` attribute from the Compose Deploy specification

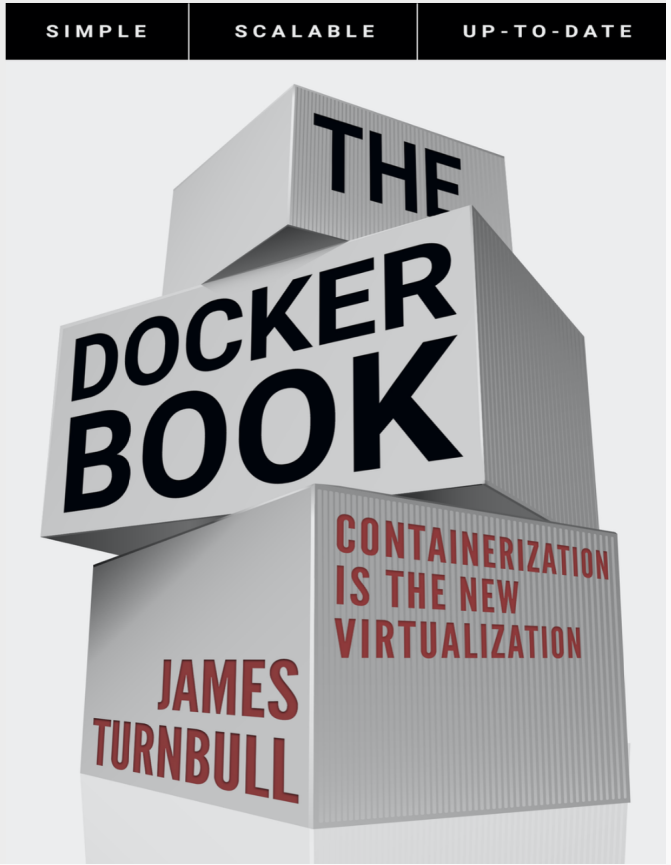
```yaml
services:
  test:
    image: tensorflow/tensorflow:latest-gpu
    command: python -c "import tensorflow as tf;tf.test.gpu_device_name()"
    deploy:
      resources:
        reservations:
          devices:
          - driver: nvidia
            device_ids: ['0', '3']
            capabilities: [gpu]
```

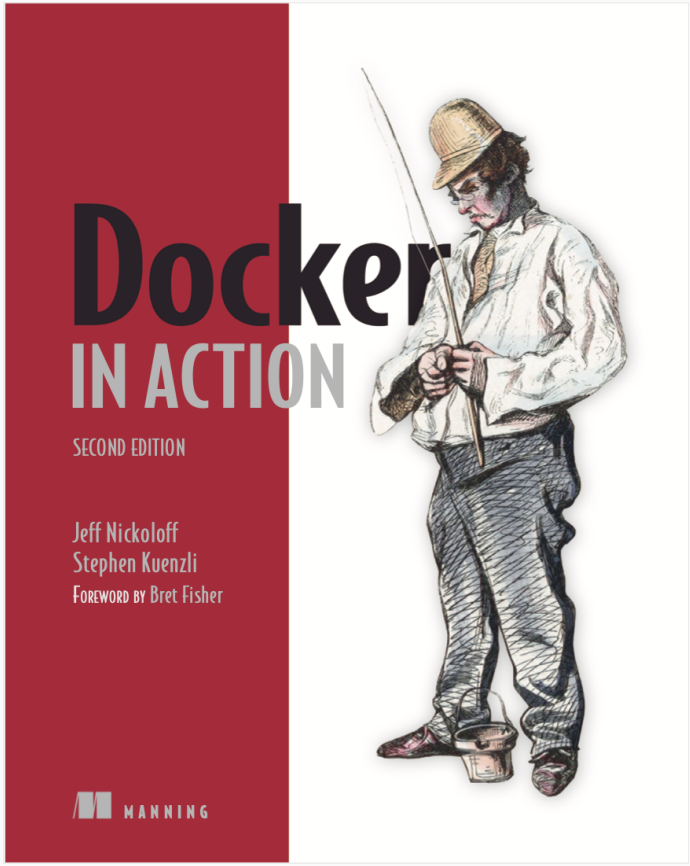
- machines with multiple GPUs: `device_ids` field can be set to target specific GPU devices

Interesting Alternatives to Docker

- SingularityCE: https://docs.sylabs.io/guides/latest/user-guide/
  - see also: Singularity: Scientific containers for mobility of compute
- Podman: https://podman.io/
- Shifter: https://shifter.readthedocs.io/en/latest/
- LXC: https://linuxcontainers.org/

Literature


SIMPLE
SCALABLE
UP-TO-DATE
THE
DOCKER
BOOK
CONTAINERIZATION
IS THE NEW
VIRTUALIZATION
JAMES
TURNBULL


Docker
IN ACTION
SECOND EDITION
Jeff Nickoloff
Stephen Kuenzli
Foreword by Bret Fisher
MANNING

References

- Slide (Virtual Environments): https://xkcd.com/1987/; 24.02.2024
- Slide (Docker Architecture): https://docs.docker.com/get-started/overview/; 24.02.2024
- Slide (Docker Containers): https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html; 24.02.2024
- Slide (What problems does Docker solve?): Jeff Nickoloff and Stephen Kuenzli - Docker in Action, Manning 2019
- Slide (Running a Docker Container): Jeff Nickoloff and Stephen Kuenzli - Docker in Action, Manning 2019
- Slide (Docker Images): Jeff Nickoloff and Stephen Kuenzli - Docker in Action, Manning 2019