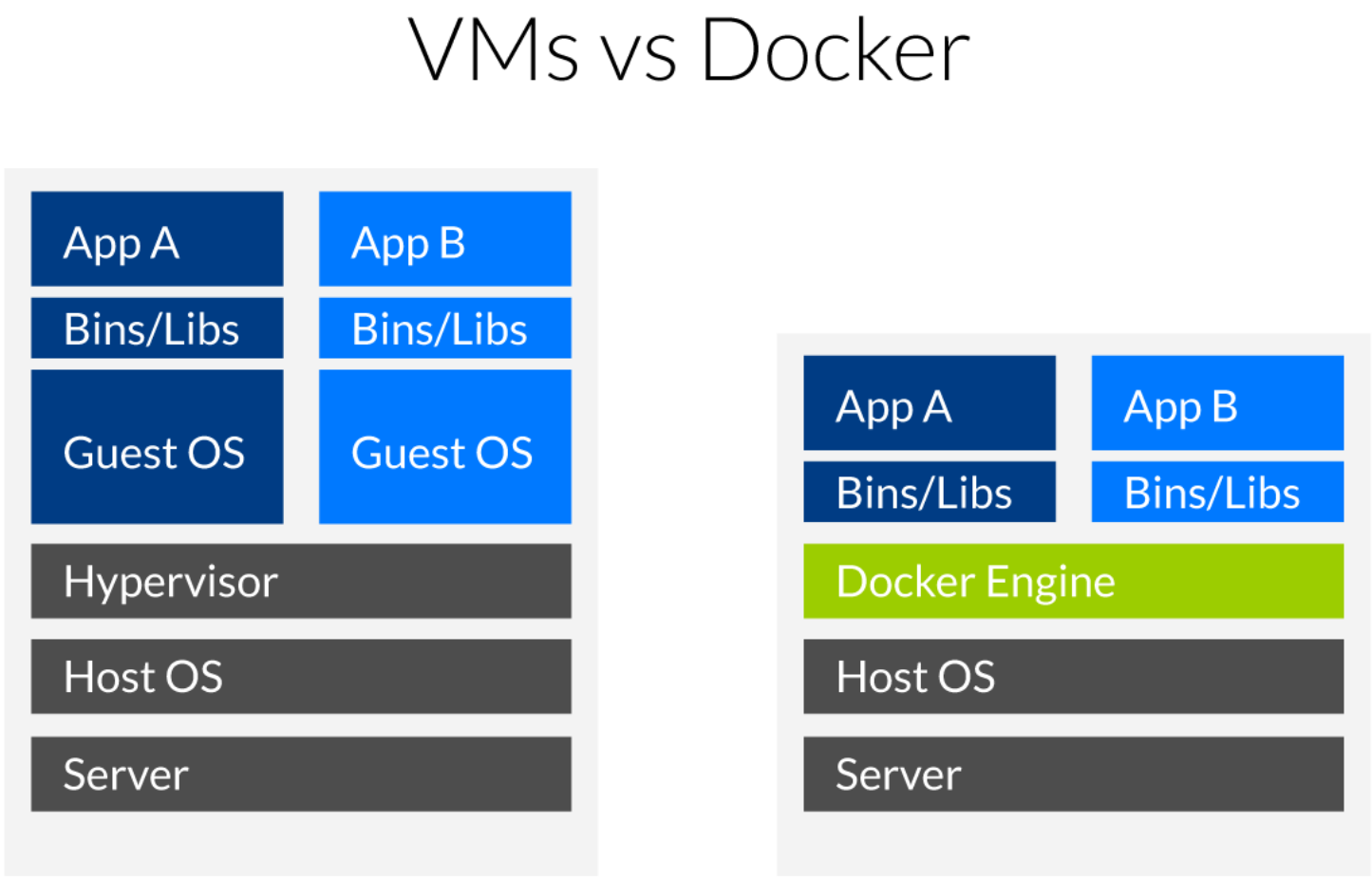
Container and provisioning

MICROSERVICES

Agenda

- Container vs. virtualization
- Linux containers
- LXC vs. Docker
- History of Docker
- Why Docker?
- Docker CLI
- Docker images
- Dockerfile
- Docker-Compose

Virtualization vs. container



Source: https://www.inovex.de/blog/docker-an-introduction-to-easy-containerization/

Virtualization vs. container

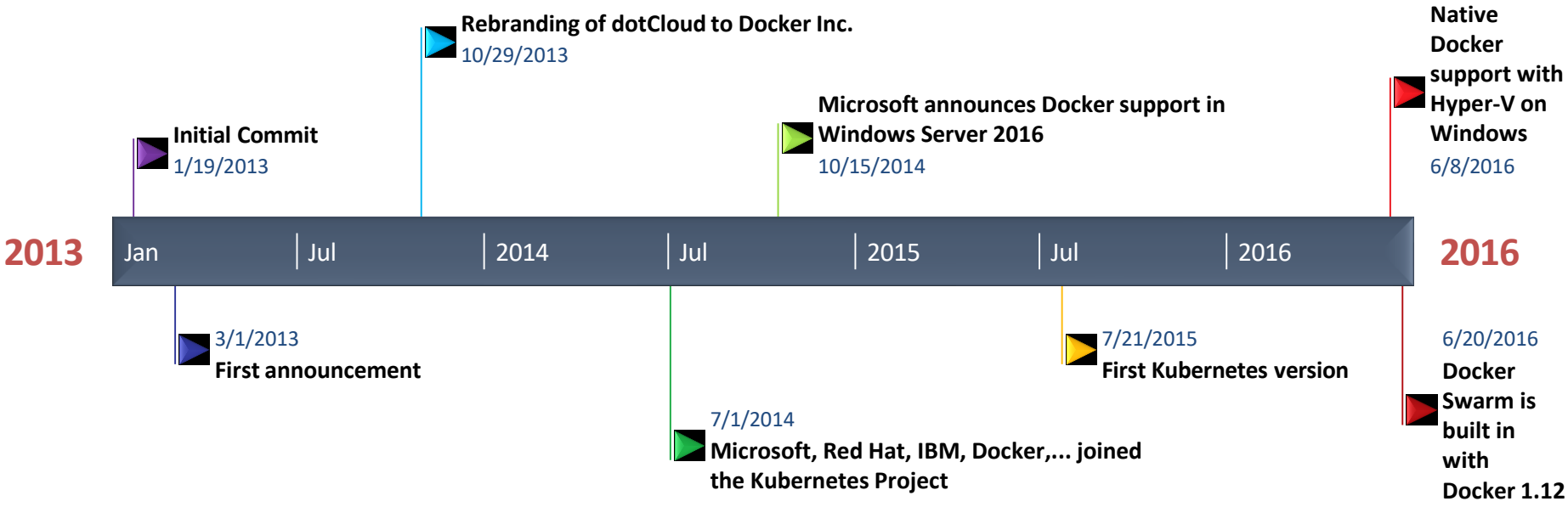
- Pros
  - Many excellent hypervisors available
  - Feels like „regular“ systems for administrators and devs
  - No special knowledge needed
- Cons
  - Resource overhead for every virtual machine
  - Takes longer to setup if no special tools are used (Puppet, Chef, etc.)

- Pros
  - Lower resource overhead
  - Very fast setup when container are prebuilt
  - Isolation of every container
  - Stable environment
- Cons
  - Training of devs and admins required
  - Getting complex if cluster is required (Kubernetes, DC/OS, Swarm)
  - Backup

Linux containers (LXC)

- Based on Linux cgroups (kernel feature to manage resources and application isolation)
- Initial release 2008
- Docker uses the same concepts as LXC! (in fact LXC is one of the available drivers for the Docker engine)

History of Docker



CONTAINERS, CONTAINERS
EVERYWHERE
memegenerator.net

Why Docker?

So after talking about pros and cons, alternative products and the history…why is Docker “the new shit”?

- No more time consuming setup of servers:
  - Dependencies
  - Configurations
  - Documentation for administrators
- Every application/component can be packed in a container
- Upgrades of containers are fast (if done right)
- Developers do not need to setup a heavy development environment but just start a view containers (Docker-Compose!)
- Administrators “just” pull the containers for production
- Containers can easily scale out (think of 5 containers of the same service instead of just 1)
- Rollback of an entire application/a single component is possible by switching the container version

Why Docker in Microservices?

- Create a container per service
- Scale out a single service instead of the whole application by deploying more containers
- Continuous integration & continuous delivery
  - E.g. automatically build new containers
  - Deploy new containers to testing, staging (and production) environments depending on which branch you‘re building
- Existing eco system for service discovery and distributed configuration (see chapter 4)
- Clustering solutions available (Docker Swarm, Kubernetes, DC/OS Mesos,…)

Docker CLI – Basics

CMDlet	Explanation
docker --help	You already guess it
docker ps	Show running docker containers
docker ps -a	Show all existing containers (including stopped)
docker images	Show all local images
docker run –ti <image[:version]>	Start a new interactive container
docker run –d <image[:version]>	Start a new container in deamon mode (in background)
docker rm <container id/name>	Removes an existing container if it is stopped
docker rm –f <container id/name>	Removes an existing container even it is still running
docker exec –ti <container id/name> /bin/bash	Attaches a Bash instance to a running container

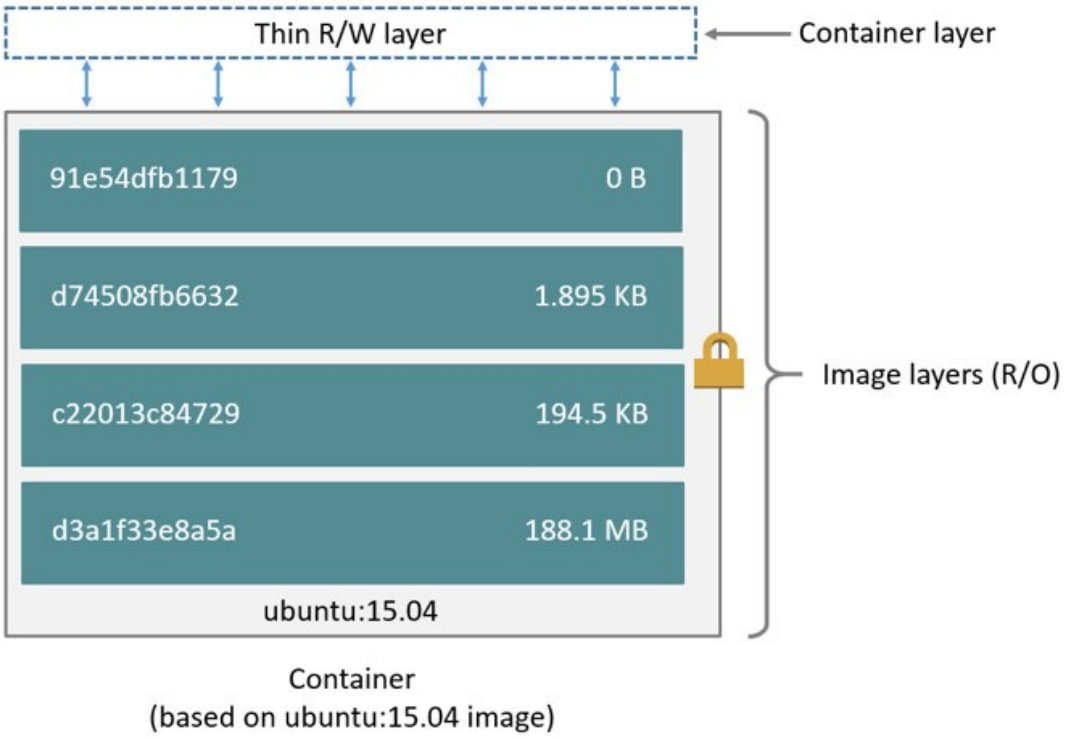
Docker CLI – Basics

CMDlet	Explanation
docker rmi <image id>	Removes a local container image
docker stop <container id/name>	Stops a running container
docker commit <container id/name> [repository[:tag]]	

Docker CLI – Build u. Registry

CMDlet	Explanation
docker build [[registry/]user/image name]	Create a new container based on a Dockerfile in the same directory
docker build [[registry/]user/image name:tag]	Create a new container based on a Dockerfile and add a tag to it
docker push [[registry/]user/image name]	Push a built image to a registry (default is Docker Hub)
docker login [registry url]	Login to a private Docker registry

Docker images



Source

Docker images

- Images consist of (more or less) layers
- Every layer is a kind of a snapshot and can be removed
- It’s possible to inspect how a specific layer was created (more on that later)
- At last each layer isn’t more than a .tar.gz archive which will be applied to the base image when a container is created
- If a image has to be rebuilt, Docker recognizes which layers aren‘t affected and is keeping them as they are to speed up the build process

Creating new Docker images

- There are two ways to create new Docker images:
  - Create a new container, do all required changes on your own by installing the via bash and *commit* this changes
  - Create a *Dockerfile*, describe all changes which have to be made to the base image and build it with the Docker CLI
- Most containers are built with Dockerfiles because it‘s easier to make small changes and recreate an image, building containers manually is only acceptable for proof-of-concepts

Dockerfile – Beispiel BeakerX

```
1    FROM beakerx-base:latest
2
3    MAINTAINER BeakerX Feedback <beakerx-feedback@twosigma.com>
4
5    ENV SHELL /bin/bash
6    ENV NB_UID 1000
7    ENV HOME /home/$NB_USER
8
9    COPY docker/setup.sh /home/beakerx
10   COPY docker/start.sh /usr/local/bin/
11   COPY docker/start-notebook.sh /usr/local/bin/
12   COPY docker/start-singleuser.sh /usr/local/bin/
13   COPY docker/jupyter_notebook_config.py /etc/jupyter/
14
15   COPY / $HOME
16
17   RUN chown -R beakerx:beakerx /home/beakerx
18
19   USER $NB_USER
20   WORKDIR $HOME
21
22   RUN /home/beakerx/setup.sh
23
24   EXPOSE 8888
25
26   CMD ["start-notebook.sh"]
27
```

Dockerfile – Basics

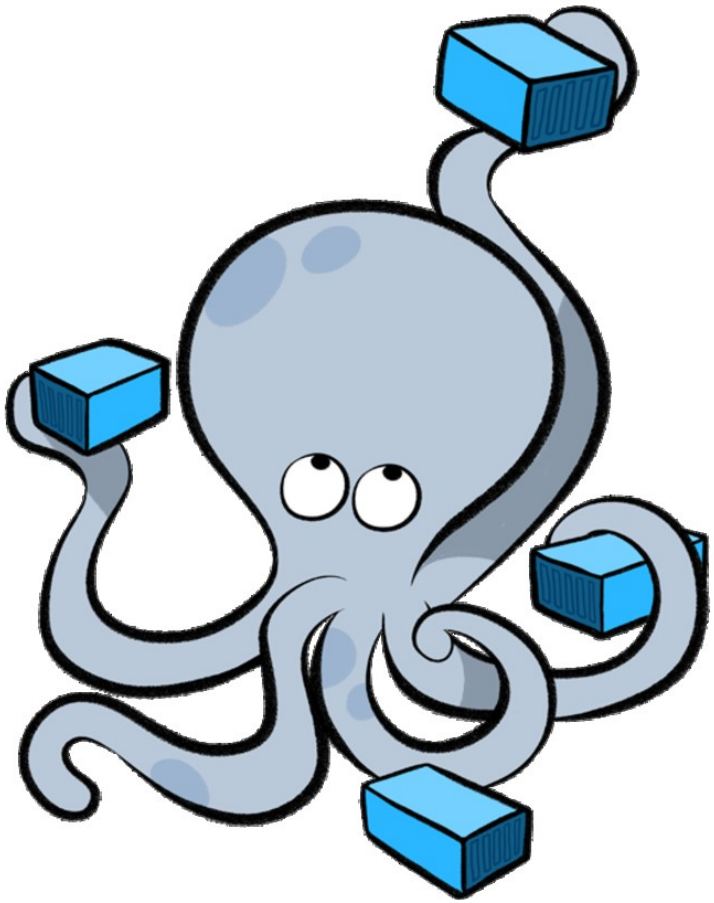
Directive	Explanation
FROM <image>[:tag]	Declares base image which will be used
RUN <command>	Command to run while building the container (creates a new layer)
CMD [„executable“, „param1“, „param2“]	Provide a default command when a new container is started
EXPOSE	Declare a port which will be exposed by the container (e.g. 80 for Nginx)
ENV <key> <value>	Declare an environment variable for the container
ADD <src> <dest>	Copy files or directories from local or remote URLs into the container
COPY	Copy files or directories from local URLs into the container
ENTRYPOINT [„executable“, „param1“, „param2“]	Declare the entrypoint of the container when it’s started

Dockerfile – Basics

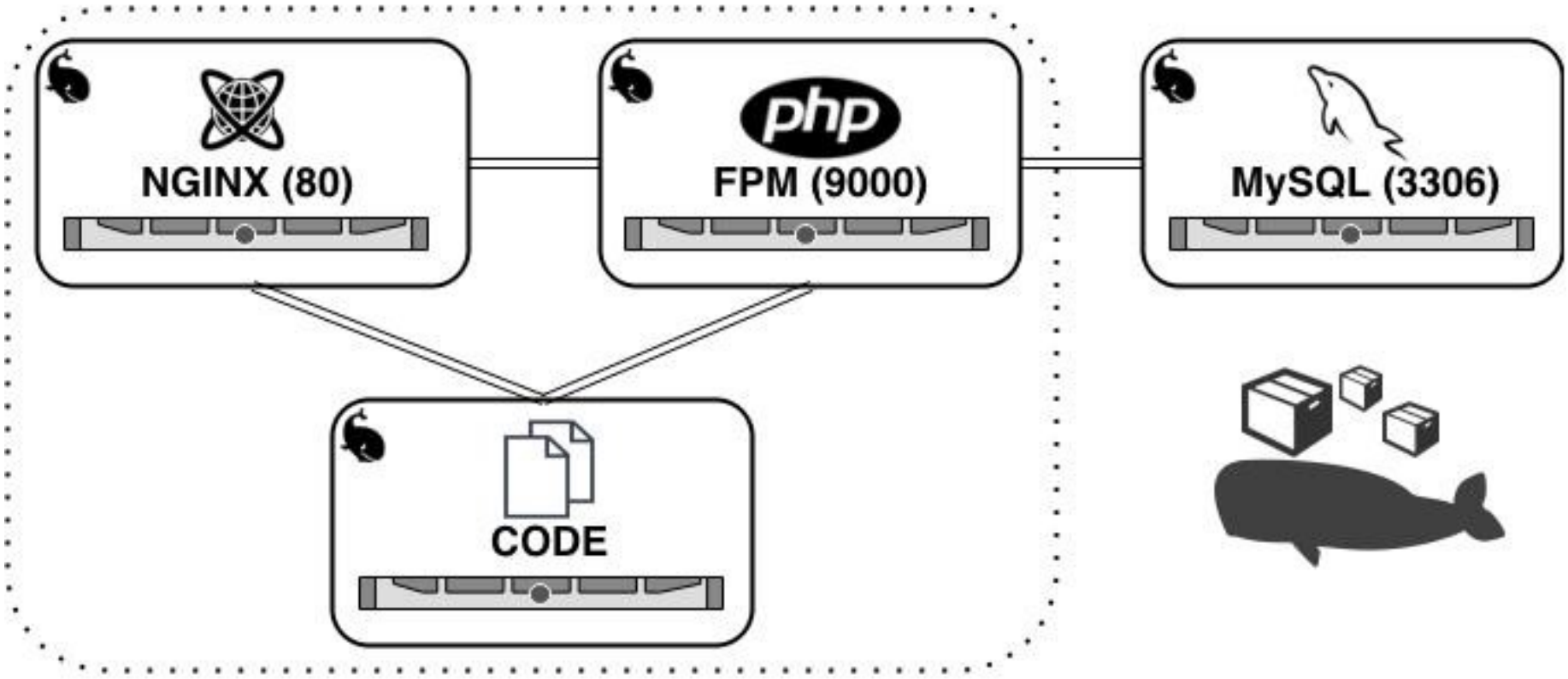
Directive	Explanation
VOLUME [“/data”]	Create a mount point to share data between the host and a container or between containers (persistence!)
USER <user>[:group]	Set the user context (and optionally the group) for all following RUN, CMD and ENTRYPOINT in the Dockerfile
WORKDIR /path/to/workdir	Sets the working directory for every following RUN, CMD, ENTRYPOINT, ADD or COPY command, can be used multiple times in one Dockerfile, the directory will be created if it does not exist, the path can also be relative

See: https://docs.docker.com/engine/reference/builder/

Docker-Compose



Docker-Compose



Source

Docker-Compose

- Tool to create multi-container applications
- Define all *services* the application consists of
- Separate *services* optionally in multiple *networks*
- Configure *services* (set environment variables, expose ports, mount volumes and so on)
- Start and stop a multi-container application by running a single command (docker-compose up/down)
- An extended version is used to deploy multi-container applications to Docker Swarm
- Other Docker cluster systems use similar formats (e.g. Pod definition in Kubernetes)
- Docs: https://docs.docker.com/compose/compose-file/compose-file-v2/
- Cheatsheet: https://devhints.io/docker-compose

Docker-Compose – Example Symfony

```
1    application:
2        image: symfony/code
3        volumes:
4            - symfony:/var/www/symfony
5            - logs/symfony:/var/www/symfony/app/logs
6        tty: true
7    db:
8        image: mysql
9        ports:
10           - 3306:3306
11       environment:
12           MYSQL_ROOT_PASSWORD: root
13           MYSQL_DATABASE: symfony
14           MYSQL_USER: root
15           MYSQL_PASSWORD: root
16   php:
17       image: symfony/php-fpm
18       expose:
19           - 9000:9000
20       volumes_from:
21           - application
22       links:
23           - db
24   nginx:
25       image: symfony/nginx
26       ports:
27           - 80:80
```