Microsevices infrastructure: Docker

Two worlds: admins vs devs

- Code sharing is easy - git push/pull
    - Repos are light - fetch only diffs
    - But each developer can and will have different environment: systems, systems versions, versions of installed libs/bins
    - Each developer can work in different projects - need to have version tools like rvm, nvm, constant changing libs versions

Two worlds: admins vs devs

- VM images sharing is hard (for developers):
  - Images are heavy
  - Builds are long
  - But libs/bins, versions and OS are same

Two worlds: what if

- What if we combine best things from this two approaches
  - Agree upon same kernel or some kind of VM
  - Build images but share only changes
  - Have simple way to create, share and manage these images
  - These images contains only layers of added thing on top of kernel

Microservices

- Each service should be as small as possible
- But each service should be isolated as possible
- Should we create VM for each of these small pieces?

Docker

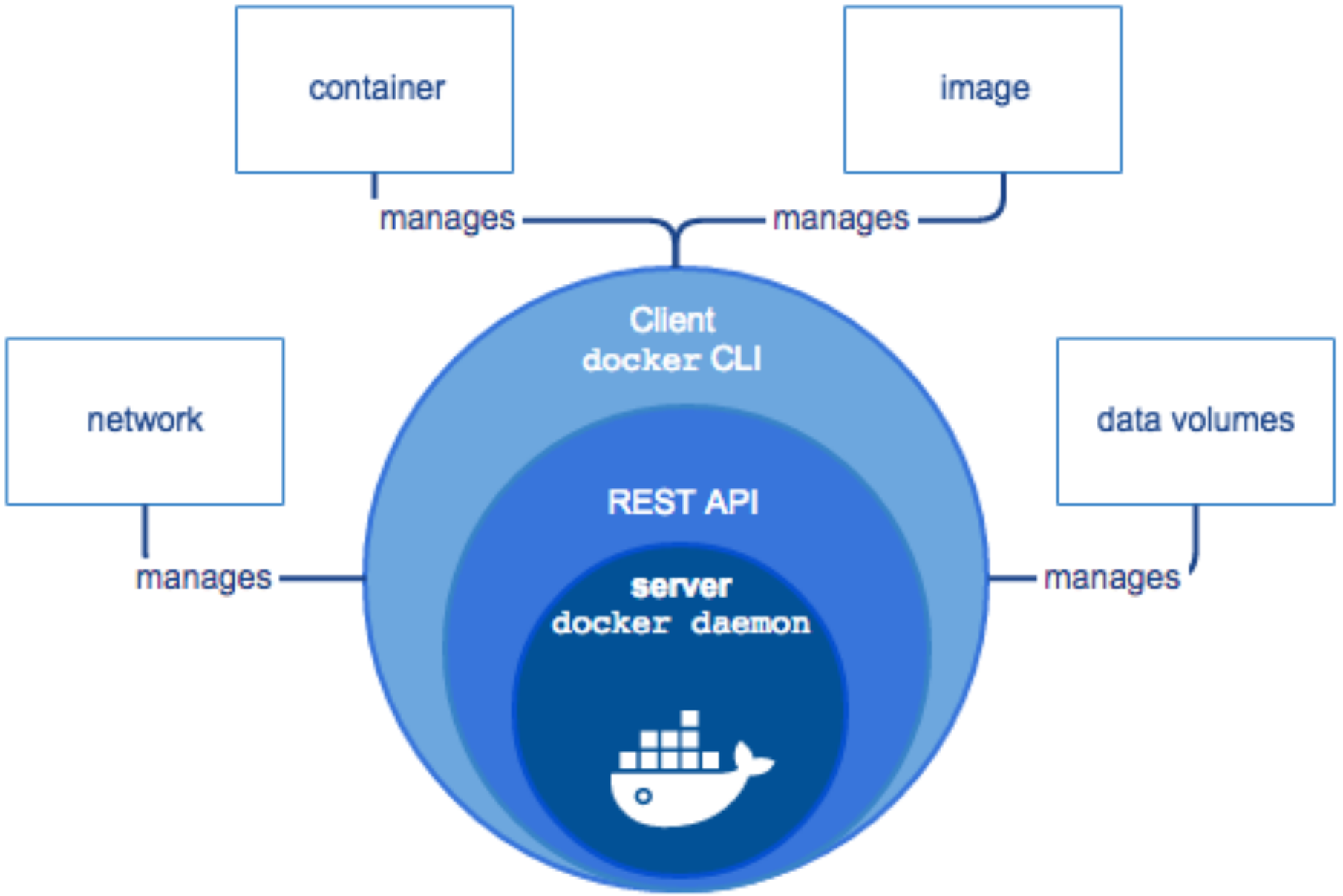
- Docker was released as open source in March 2013.

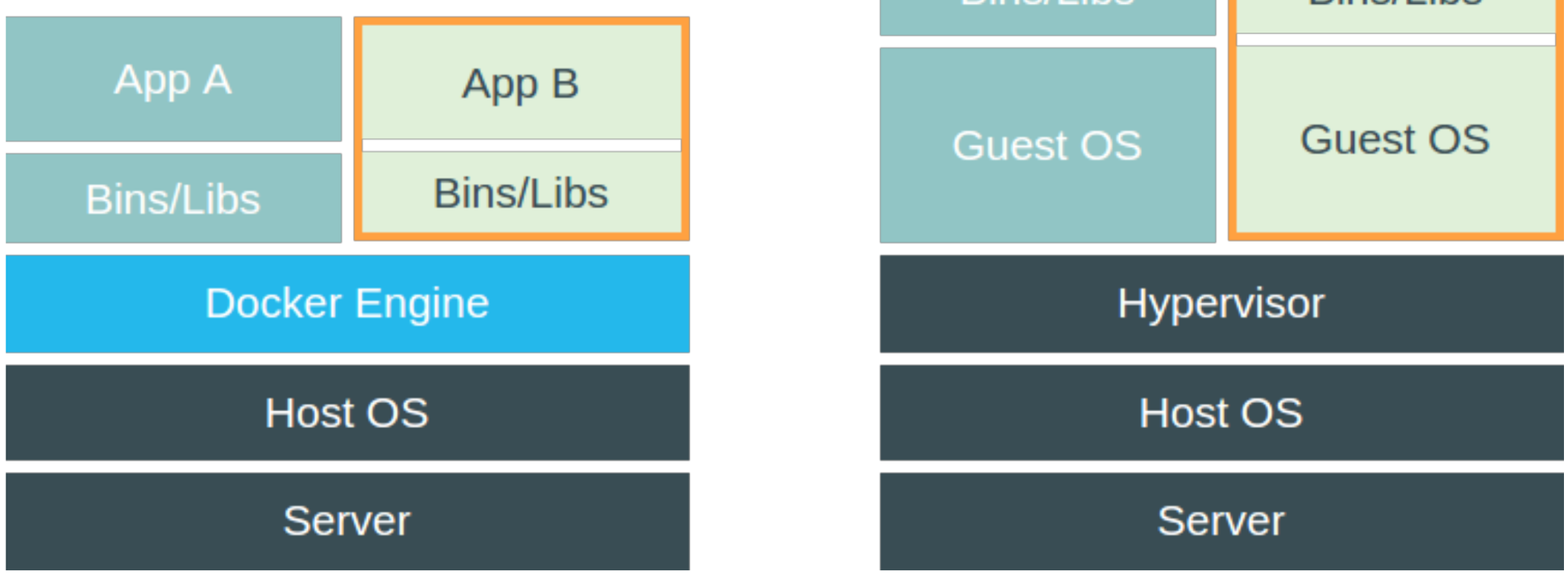
- March 2014, with the release of version 0.9, Docker dropped LXC as the default execution environment and replaced it with its own libcontainer library written in the Go.

- Docker uses the resource isolation features of the Linux kernel such as cgroups and kernel namespaces, and a union-capable file systems UnionFS

Docker: Secure by Default.

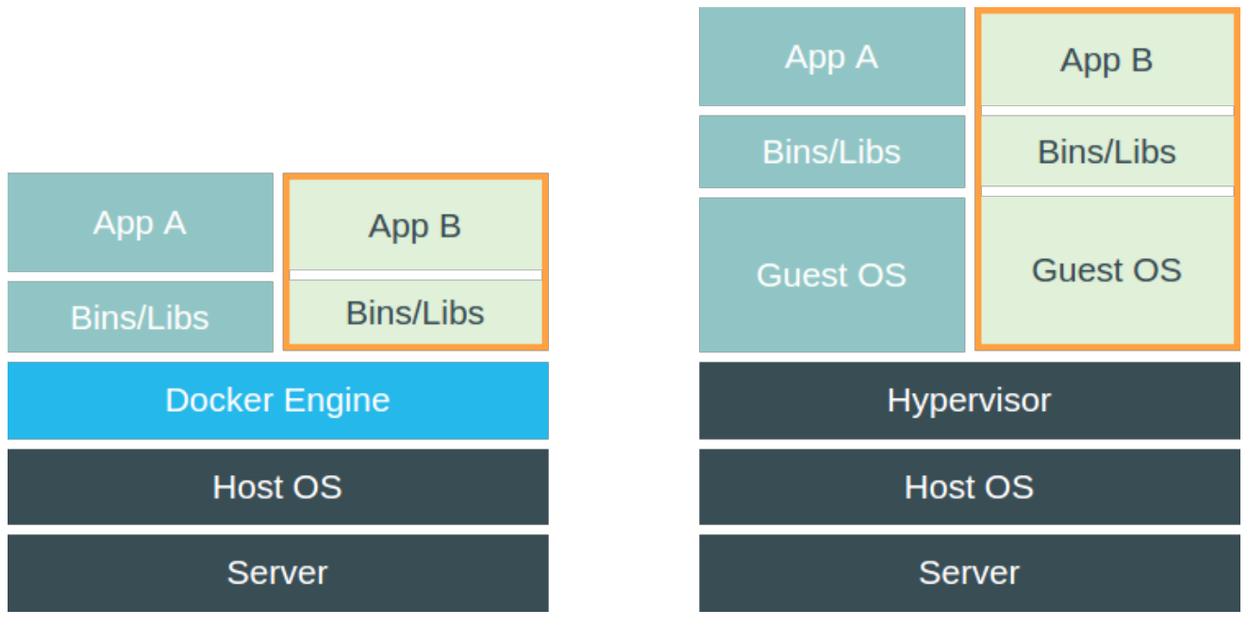
- Each container has own:
    - Namespace: pid, uid, itc, mnt, net
    - Cpgroup
- Each container has enabled:
    - AppArmor policy


container
image
manages
manages
Client
docker CLI
REST API
server
docker daemon
network
data volumes
manages
manages



containers are not vm

They share same kernel with host



- Less isolated - but isolated from each other
- Light-weighted
- Sharable

Approaches

- Scary monolith
- Monolith with services
- Microservices

Scary monolith: bad approach

- Use only base images
- Exec (or even ssh) to container, config by hand
- Run all needed processes in one container:
    - application, db, cache-store, background-processing, etc
- Container dies - config by hand once again

Scary monolith: better approach

- Proper dockerfile
- docker pull; docker run
- All processes in one container
- example: Gitlab omnibus

Monolith with services

- Service is a set of processes needed to fulfill some requirements, eq:
  - App server, web server background processing.
  - Db
  - Cache-store

Monolith with services: real world example

- Web server
- App server
- Real-time server
- Background processing
- Scheduler
- Database
- Cache-store
- Message bus

Monolith with services: real world example

- And for local development:
  - Local mail server
  - Webpack
  - http tunnel (ngrok)

Monolith with services

Good way to start writing more **micro**services

Mircoservices

- Each container run one (group - eq unicorn) process
- Easy to inspect and reason about behavior
- Hard to build by hand - **docker-compose** for rescue

Docker

Docker

- Dockerfile
- Docker commands
- Docker run
- Docker-compose
- DevOps
- Threats
- Recommendations

Dockerfile

Dockerfile

- Contains directives to create docker image
- Each directive creates another image layer
- All created layers are read-only
- FROM, ENV, ADD, COPY, CMD, RUN

Docker commands

Docker commands

- Docker images
- Docker volumes (create ls)
- Docker pull
- Docker ps (-a -s)
- Docker logs / attach / stats
- Docker diff

Docker commands

- Docker run
- Docker exec (-it)
- Docker start, stop
- Docker kill
- Docker rm

Docker run

- -d —name
- -rm
- -p host:container
- —read-only
- -mount source=named-volume,target=path-in-conatiner
- -v source:target (/tmp:/app/logs)

Docker run

- —pid=host - drop namespace capability
- —cgroup-parent
- —memory —memory-swap
- —cpus —cpu-shares (default 1024) —cpu-quota (0 - 100)
- —blkio-weight (10 to 1000 default 500)
- —cap-add —cap-drop
- —oom-kill-disable

Default capabilities

Capability Key	Capability Description
SETPCAP	Modify process capabilities.
MKNOD	Create special files using mknod(2).
AUDIT_WRITE	Write records to kernel auditing log.
CHOWN	Make arbitrary changes to file UIDs and GIDs (see chown(2)).
NET_RAW	Use RAW and PACKET sockets.
DAC_OVERRIDE	Bypass file read, write, and execute permission checks.
FOWNER	Bypass permission checks on operations that normally require the file system UID of the process to match the UID of the file.
FSETID	Don't clear set-user-ID and set-group-ID permission bits when a file is modified.
KILL	Bypass permission checks for sending signals.
SETGID	Make arbitrary manipulations of process GIDs and supplementary GID list.
SETUID	Make arbitrary manipulations of process UIDs.
NET_BIND_SERVICE	Bind a socket to internet domain privileged ports (port numbers less than 1024).
SYS_CHROOT	Use chroot(2), change root directory.
SETFCAP	Set file capabilities.

Docker-compose

Docker-compose

- Running all needed containers by hand is hard
- We can write bash scripts
- But we can use docker-compose

Docker-compose

- Define YAML file
- Declarative style
- Run docker-compose up/ down/stop/restart/run

```
version: '3'
services:
  web:
    build: .
    ports:
    - "5000:5000"
    volumes:
    - .:/code
    - logvolume01:/var/log
    links:
    - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

DevOps

DevOps - typical pipeline

- Build image
- Run tests
- Upload image to registry
- Deploy (on many hosts if necessary)
- Pull as developer

DevOps - debugging

- Fetch prod docker-compose
- Pull prod image
- docker-compose run

DevOps - feature apps

- We want to deliver with agile style
- With master-staging branch developers can blocks each other
- What if each feature branch could have own staging?

DevOps - feature apps

- With docker/docker-compose - simple as 🥧
    - Build image
    - pull in staging VM,
    - docker-compose run
    - nginx-proxy for magical host discovery

Threats

Threats

- Poisoned images
- Kernel vulnerabilities
- Container takeover:
    - Container breakout
    - Secrets leakage
    - Neighbors sniffing
- DoS
- Unrestricted access to REST API
- Unrestricted access to docker / root group on host

Recommendations

General recommendations

- Limit access to docker host
- Keep Docker deamon and kernel up to date
- Mix two world - VMs and containers
- Keep services with critical data/accesess away from regular one (on different machine or different VM)
- least privilege, least access
- Write your own Seccomp polices and use them

Dockerfile recomendations

- Carefully chose base image
    - Do not forget about default config
- If possible - tailor your own, based on minimalistic image
- Change USER
- One processes per container
    - No deamons
    - Print logs to stdout

Dockerfile recommendations

- Docker Security Scanning
- CorsOs - Clair
- Docker-bench

Docker run recommendations

- Believe in default config
- Bind ports if necessary
- Never run —privileged option
- Do not run in default network
- Drop all unused capabilities
- Limit resources
- Use namespaces

Docker run recommendations

- Run —read-only if possible
- Be careful with volumes
    - Do not mount to sensitive dirs
    - If you must - read only
- Create as many network as needed

Running container recommendations

- Central logs / monitoring (ELK stack, papiertrail, grafana)
- Immutable infrastructure
  - All containers with same image share same behavior
  - Audit images - not containers
  - Do not be afraid of killing

Secrets recommendation

- Use secrets per container
- options:
  - Bad: store in Dockerfile
  - Better: pass in env variables
  - Better: mount volume
  - Better: Use Vault (maybe in container)

REST API recommendation

- Do not mount /var/run/docker.sock to container
- Hide
- Force VPN