loodse

Siemens Day I
18.03.2019

Container Fundamentals

loodse

Who is?

loodse

Operational Excellence for Your Cloud Native Applications

Our Expertise

Loodse is a leading expert for container and cloud native technologies.

Our Team

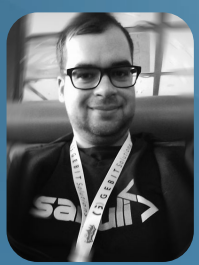
We are a team of 35+ employees across Europe and the US.

Locations

Founded in 2016, our headquarters are located in Hamburg, Germany.

kubernetes CERTIFIED SERVICE PROVIDER

kubernetes TRAINING PARTNER



Tobias Schneck

tobi@loodse.com

toschneck

@toschneck

Who are you?

What is your background?
What is your job?
What is your expectation?

A helper:

Please name three things like:

- a tool/language/technology you work with
- something you really love or hate
- your work tasks or challenges
- something you want to learn

loodse

Rules & Agenda

Rules

1. Please interrupt me! Anytime, really!
   This needs to be interactive or we are wasting our times.

2. Have an idea for something to talk about? Let me know.
   If you really do not want to interrupt me for it, you can just send me an e-mail too.

3. If you are stuck in hands-on or could not follow what I was saying:
   → Rule 1.

4. Please try to keep track of what I am talking, because when I lose the thread, you really need to help me get back to it.

loodse

Agenda

Container Fundamentals
- Why Containers?
- Architecture of Containers:
- Linux, Namespaces, cgroups
- What is Docker?
- Dev/Ops Impact

Container Hands-On
- Installation
- Images
- Container:
- Handling and Interaction
- Dockerfiles
- Image Builds
- Advanced Dockerfiles
- Networking
- Volumes
- Tipps & Tricks
- Labs

loodse

http://bit.ly/loodse-siemens-03-2019

Open for Training Material

loodse

Lesson #1
A quick not-so-technical start...

loodse

Why the Container Hype?

The software industry is going through a transition

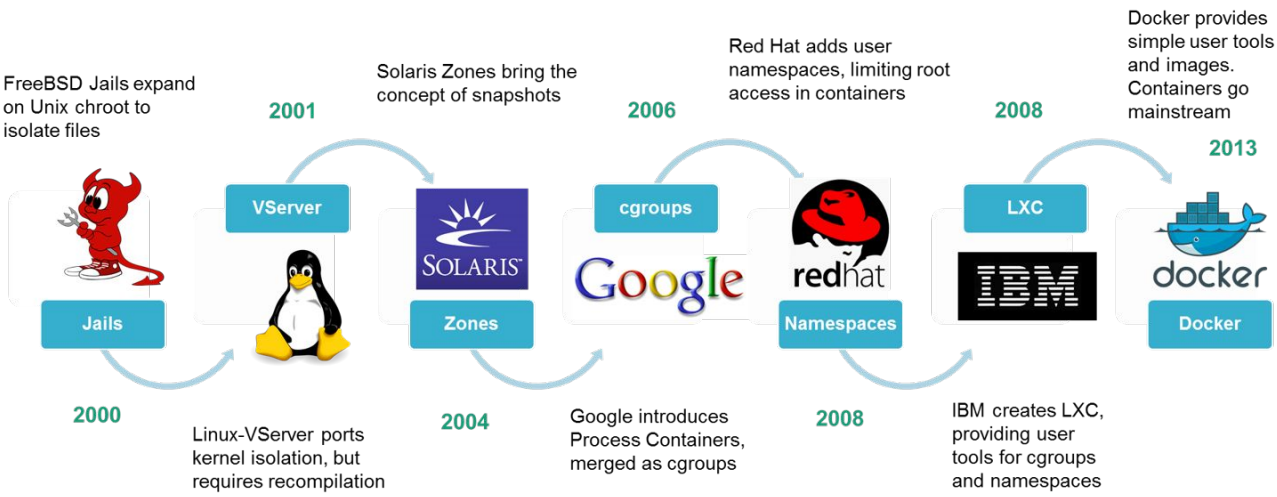
- From:
  - Monolithic applications
  - Long development cycles
  - Environment-dependency
  - Low scalability
- To:
  - Distributed applications
  - Fast, iterative improvements
  - Multiple, isolated environments
  - Faster scalability

loodse

The Deployment Challenge

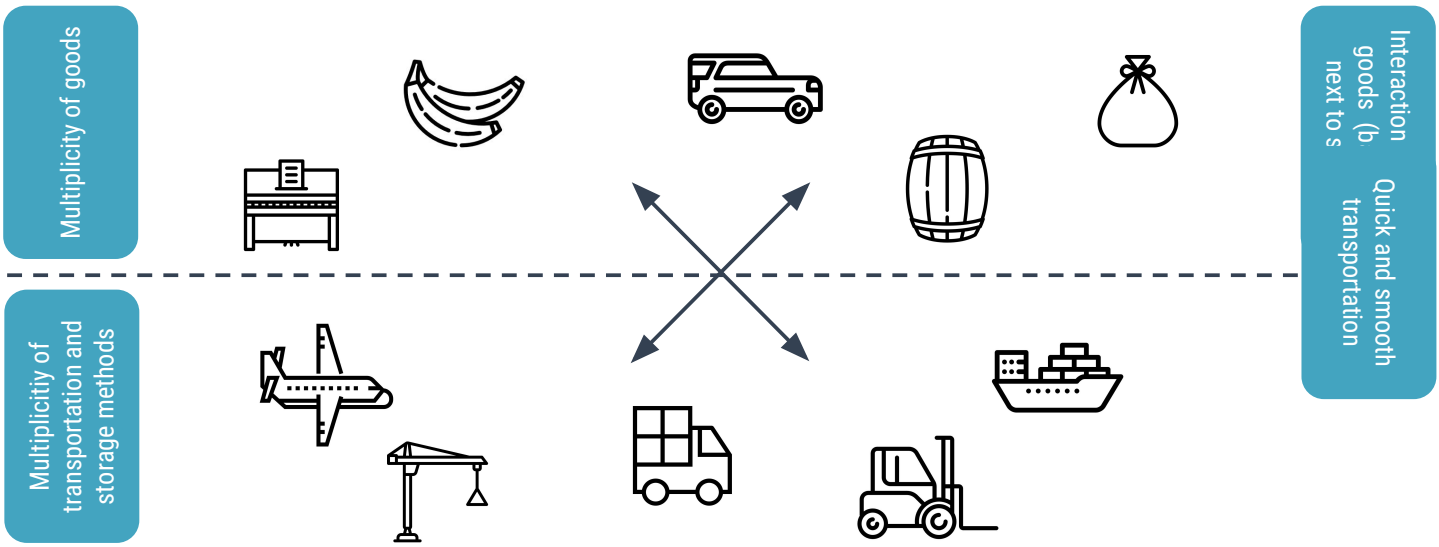
- Many different stacks:
  - Languages
  - Frameworks
  - Databases
- Numerous targets:
  - Individual development environments
  - Pre-production, QA, staging...
  - Production: on premise, cloud, hybrid

A Timeline



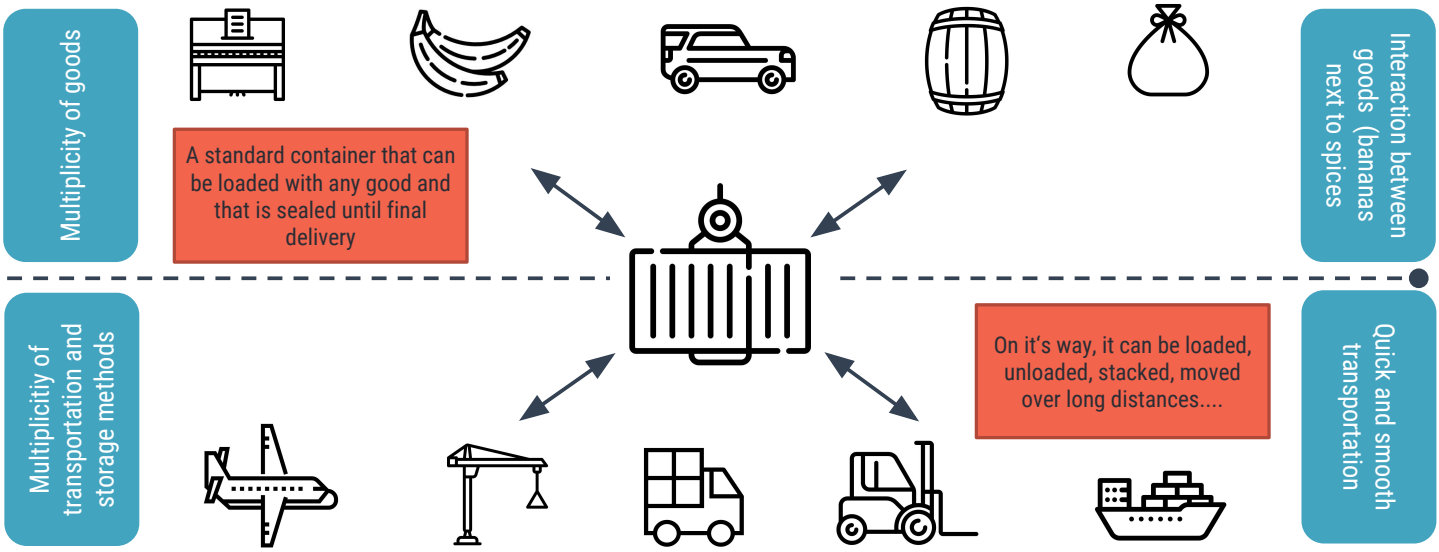
loodse

Bananas and Coffee



loodse

The Rise of Shipping Containers



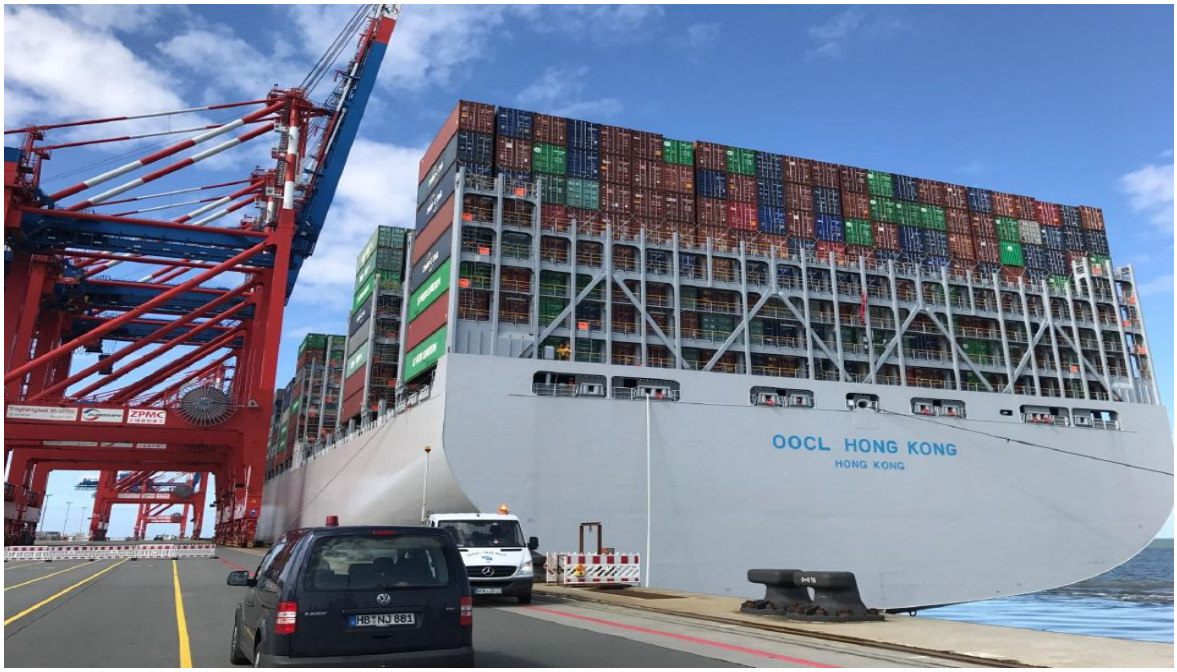
loodse

USP: storage and transport tool

loodse

Handling and stackability

Standardization



Flexibility

Container Types

Standard Container

High-Cube Container

Hardtop Container

Open Top Container

Tank Container

Flat

Platform (Plat)

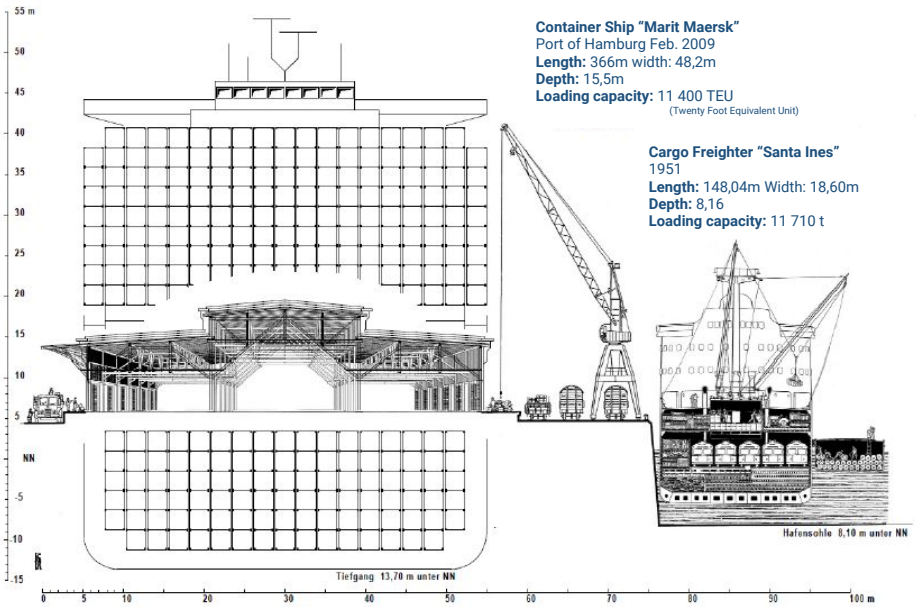
Ventilated Container

Cooling Container

Bulk Container

Logistic chain

General cargo freighter vs. container ship



Container Ship "Marit Maersk"
Port of Hamburg Feb. 2009
Length: 366m width: 48,2m
Depth: 15,5m
Loading capacity: 11 400 TEU
(Twenty Foot Equivalent Unit)

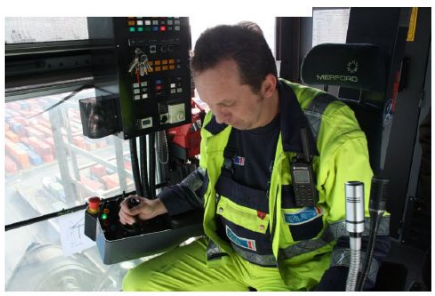
Cargo Freighter "Santa Ines"
1951
Length: 148,04m Width: 18,60m
Depth: 8,16
Loading capacity: 11 710 t

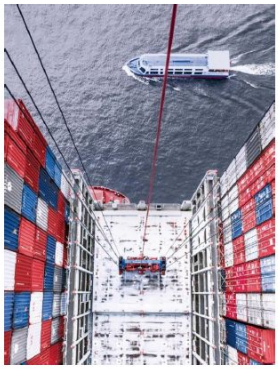
loodse

What working in the port used to look like...

loodse

...and what it looks like now:





The Shipping Container Ecosystem

Same thing!

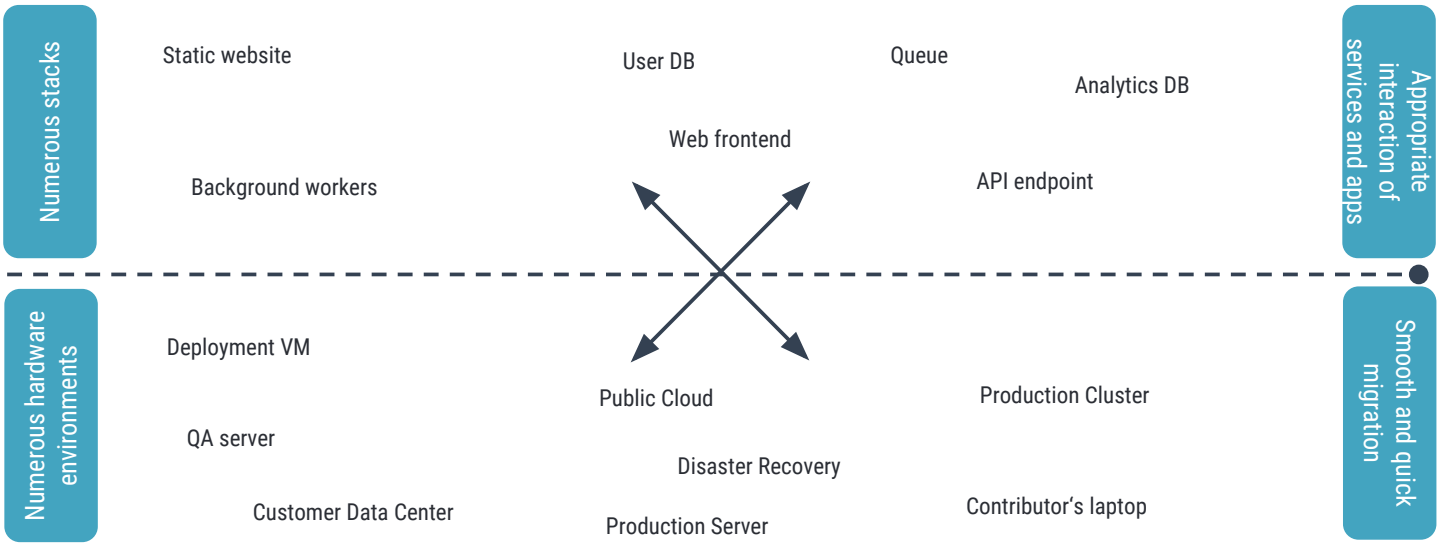
BASH
THE BOURNE-AGAIN SHELL

```
#!/bin/bash
```

Java™

loodse

The Deployment Challenge



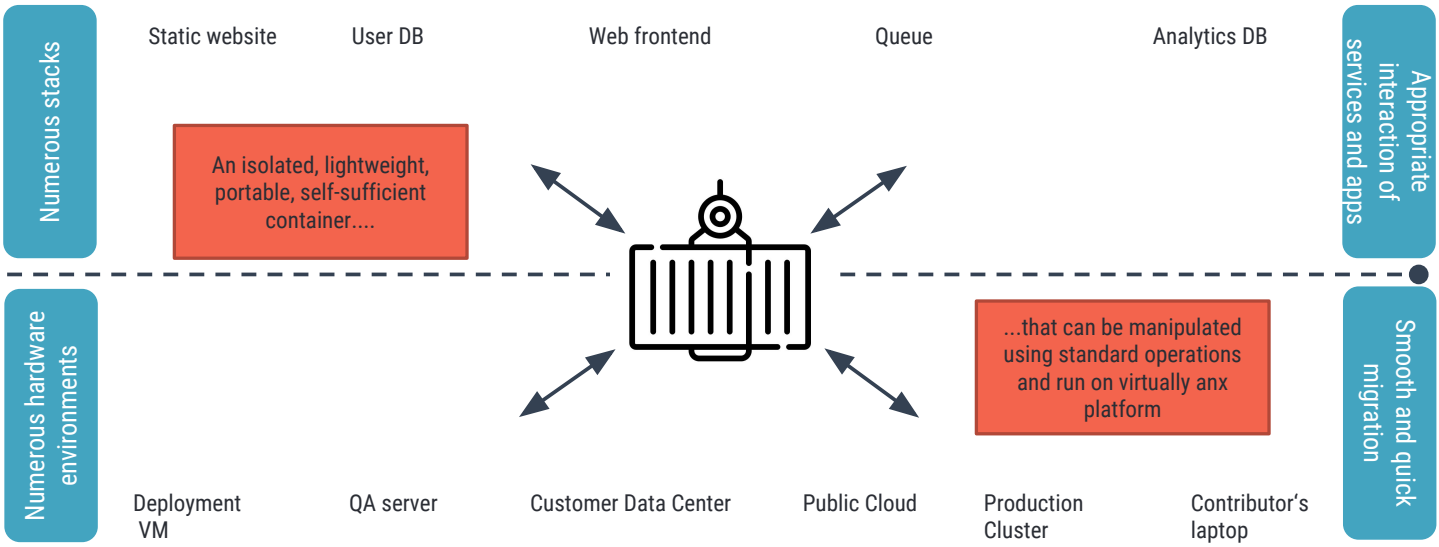
loodse

A Nightmare

	Development VM	QA server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's labtop	Customer Servers
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

loodse

A Container System for Applications



loodse

Escape the Nightmare

	Development VM	QA server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's labtop	Customer Servers
Static website							
Web frontend							
Background workers							
User DB							
Analytics DB							
Queue							

Outcome

- Standardization
  - Reduced dev-to-prod time
  - Reduced continuous integration job time
- Declarative
  - leads to reproducibility
- Abstraction and Isolation
  - contracts
  - Independence
- Ease
  - Lots of containers!

loodse

What is a container?

Really, in technical terms, what is it?

loodse

Namespaces

Tasks:

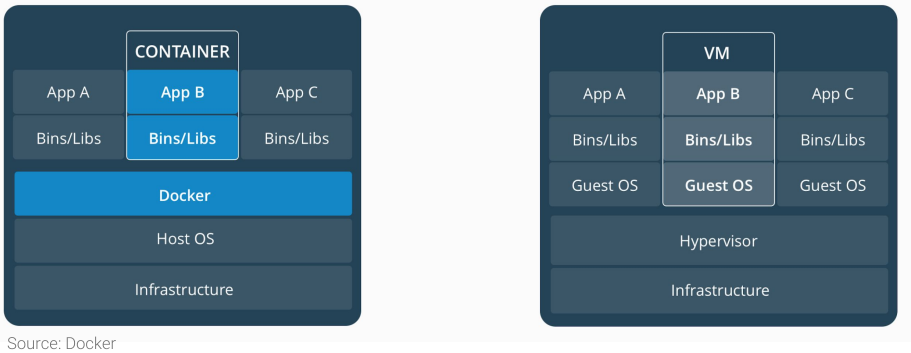
- Create a process which has a different hostname than its parent
- Create an isolated network ns and communicate with the parent (bonus available)
- Create a new pid ns and show the namespaces processes

loodse

Containers vs VMs

Containers are isolated but share OS kernel and, if appropriate, bins/libraries

- Faster deployment
- Less overhead
- Easier migration
- Faster restart



Source: Docker

Escape the “Worked in dev - ops problem now!“

1. You have that "`INSTALL.txt`" file in human language?
2. Take the file and write an "`install.sh`" from it!
3. Turn this file into a "`Dockerfile`“
4. Test it
5. If it builds on your machine, it will build anywhere (standardized environment)

Interact with Containers

- create, run,
- start, stop,
- exec,
- attach,
- tag,
- build,
- pull/push,
- login,
- commit,
- diff

loodse

Docker images

- Docker images
  - Layers
  - Deduplication
- Interactive building
  - Filesystems
- Dockerfile
  - declarative

loodse

Task: Create a counter

⇒ Next: in a programming language (!= shell)

What is Docker?

loodse

Formats and APIs before

- No standardized exchange format
- No portability of containers
- No re-usable components, APIs, tools. (At best: VM abstractions, e.g. libvirt.)

=> Containers were hard to use for devs and hidden from the end users (no equivalent of docker run debian)

loodse

Formats and APIs with Docker

- Standardized container formats to make them portable
- Ease-of-use for developers
- Focus on re-usable components, APIs and standard tools
- Improvement over ad-hoc, in-house, specific tools

loodse

Shipping before

- Ship packages: deb, rpm, gem, jar, homebrew...
- Dependency nightmare
- "Works on my machine."
- Unreliable base deployment, often done from scratch (debootstrap...)

Shipping with Docker

- Ship container images with their dependencies
- Bigger images, but broken down into layers
- Only ship layers that have been modified
- Save disk, network and memory usage

Layers

Examples:

- CentOS
- JRE
- Tomcat
- Dependencies
- Application JAR
- Configuration

Breaking Down the Dev/Ops Divide

The Devs/Ops Divide

- Dev environment is very different from production
- Ops often do not have a dev environment themselves
  ⇒ When they do, it can differ from the devs' environment
- Ops have to identify differences and make it work
  ⇒ If they can't: bounce it back to devs
- Shipping code causes frictions and delays

Devs/Ops with Docker

- Drop a container image file (or a Docker Compose / Kubernetes manifest)
- Ops can run that container with predefined deployment
- Ops still have to adapt to prod environment, but they now have a point of reference
- Ops have tools enabling them to use the same image in dev and prod
- Devs can be empowered to make releases themselves more easily

loodse

Clean Separation of Competencies

The Devs side

Everything „inside“ the container:

- Code
- Libraries
- Package Manager
- Apps
- Data

Note: All Linux containers look the same

The Ops side

Everything „outside“ the container:

- Logging
- Remote access
- Monitoring
- Network config

Note: All containers start, stop, copy, attach etc. the same way

loodse

The Origins of Docker

- dotCloud was operating a PaaS, using a custom container engine based on OpenVZ and AUFS
- It started around 2008 as a single Python script
- By 2012, the engine had about 10 Python components and 100 other microservices
- End of 2012, dotCloud refactors the engine under the project codename „Docker“
- In 2013, "Docker" is released with an open source license
- In 2014, the PaaS activity is sold

loodse

Early Adopters

- PaaS builders
- Big PaaS users
- CI platforms
- devs

⇒ Maturity (2015-2016)

Docker as Industry Standard

- Docker reaches 1.0 milestone
- Existing systems like Mesos and Cloud Foundry integrate Docker support
- Standards like Cloud Native Computing Foundation (CNCF) and Open Containers Initiative (OCI) appear
- Other container engines appear, e.g. Rocket (core os)

loodse

Setup Katacoda > Playground

https://www.katacoda.com/loodse/courses/docker

Lesson #2
Getting Started with Docker

loodse

Installing Docker

Docker is easy to install. It runs on:

- A variety of Linux distributions
- OS X via a VM
- Microsoft Windows via a VM

loodse

Installing Docker on Linux

It can be installed via:

- Distribution-supplied packages on almost all distros (including at least: Arch Linux, CentOS, ~~Debian~~[1], Fedora, Gentoo, openSUSE, RHEL, Ubuntu)
- Packages supplied by Docker
- Installation script from Docker
- Binary download from Docker

[1] not for Debian 9 - Stretch (but arrived in testing…)

loodse

Installing Docker with Upstream Packages

- Preferred method for Linux and more up-to-date than distros'.
- Instructions per distro: https://docs.docker.com/engine/installation/linux/
- Package will be named docker-engine.

loodse

Installing Docker with Distros Packages

On Red Hat derivatives (Fedora, CentOS):

```
$ sudo yum install docker
```

On Debian and derivatives:

```
$ sudo apt-get install docker.io
```

loodse

Installation Script from Docker

curl command can be used on several platforms:

```
$ curl -s https://get.docker.com/ | sudo sh
```

This currently works on:

- Ubuntu
- Debian
- Fedora
- Gentoo

loodse

Installing on OS X and Microsoft Windows

Docker doesn't run natively on OS X or Microsoft Windows* but there are three ways to make it work:

- Use Docker for Mac or Docker for Windows (recommended)
- Use the Docker Toolbox (deprecated)
- Roll your own with e.g. Parallels, VirtualBox, VMware...

loodse

Running Docker on OS X and Windows

When you execute docker version from the terminal:

- The CLI connects to the Docker Engine over a standard socket
- The Docker Engine is actually running in a VM but the CLI is not aware of that
- The CLI sends a request with REST API,
- The Docker Engine in the VM processes the request
- The CLI gets the response and shows it to you

Note: All communication with the Docker Engine goes over the API. This allows you to use remote engines in exactly the same way as local engines

Rolling your own Install

- Think at least twice about it. There is almost no good reason to do that.
- If you need something very custom, the Docker Eco is probably the better choice

Using the Docker Toolbox

The Docker Toolbox installs the following components:

- VirtualBox + Boot2Docker VM image (runs Docker Engine)
- Kitematic GUI
- Docker CLI
- Docker Machine
- Docker Compose
- A handful of clever wrappers

About boot2docker

- It is a very small VM image (~30 MB) (but not a „lite“ version of Docker)
- It runs on most hypervisors
- It can boot on actual hardware

loodse

Docker Mac and Docker Windows

- Docker Mac and Docker Windows allow to run Docker without VirtualBox
- They are installed like normal apps
- They provide better integration with enterprise VPNs
- They support filesystem sharing through volume

Note: For now, they only run one instance at a time. If you want to run a full cluster on your local machine, you can fallback on the Docker Toolbox

Important PSA about Security

- The docker user is root equivalent. It provides root-level access to the host.
- You should limit access to it like you would protect root
- If you give somebody access to the Docker API, you give him full access on the machine
- To avoid unauthorized access on multi-user machines, the Docker control socket is by default owned by the Docker group
- If a user is not member of the Docker group, you need to prefix every command with sudo

loodse

The Docker Group

Add the Docker group

```
$ sudo groupadd docker
```

Add ourselves to the group

```
$ sudo gpasswd -a $USER docker
```

Restart the Docker daemon

```
$ sudo service docker restart
```

Log out

```
$ exit
```

Docker as Central Platform

- The initial container engine is today known as "Docker Engine“
- Adding of more tools:
  - Docker Compose (formerly "Fig")
  - Docker Machine
  - Docker Swarm
  - Kitematic (Generic UI)
  - Docker Cloud (formerly "Tutum")
  - And more
- Docker Inc. launches commercial offers
  - Docker EE (formerly Datacenter)

loodse

Lab > Install Docker

https://www.katacoda.com/loodse/courses/docker

loodse

Lesson #3
Our first container

loodse

Docker Set-Up

Docker is a client-server application

- The Docker Engine (or "daemon")
- The Docker Client
- Container Registry
  - http://hub.docker.com/
  - http://store.docker.com/

loodse

Hello World

In your Docker environment, simply run this command:

```
$ docker run alpine echo hello world
```

loodse

That Was Already It - Our Very First Container!

- `$ docker run alpine echo hello world`
- `Run` a command in a new container
- `alpine` is one of the smallest Linux systems
- `echo hello world` is the command which is executed

loodse

Docker Run

- `Docker run` processes in isolated containers.
- A container is a process which runs on a host.
- The host may be local or remote.

⇨ When an operator executes docker run, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host.

loodse

Run an Apache in a Container

```
$ docker run -it –p 80:80 debian
```

For interactive processes (like a shell), you must use `-i -t` together in order to allocate a `tty` for the container process. `-i -t` is often written `-it`.

To expose a container's internal port, you can start the container with the `-P` or `-p` flag. The exposed port is accessible on the host and the ports are available to any client that can reach the host.

loodse

Install a Apache

Now, let's install it:

```
root@2bc0dd255339:/# apt-get update && apt-get install -y apache2
```

loodse

Install a Apache

Start the Apache:

```
root@2bc0dd255339:/# apache2ctl -DFOREGROUND
```

loodse

Open the Website

Go to http://ip_adress_of_your_machine

loodse

Stop the Process and Exit the Container

Exit the process with (E.g. with ^C) as you normally would do

Exit the shell with ^D or type `exit`

```
root@2bc0dd255339:/# exit
```

The container is stopped but it still exists on disk. You can take a look with the `docker ps` command

```
$ docker ps -a
```

Lab > Our first container

https://www.katacoda.com/loodse/courses/docker

loodse

Lesson #4
Run a Webapp with Docker

Foreground Container

In foreground mode (by default when `-d` is not specified), `docker run` can start the process in the container and attach the console to the process' standard input, output, and standard error.

It can even pretend to be a `TTY` (this is what most command line executables expect) and pass along signals.

```
$ docker run -t -p 80:80 loodse/demo-www
```

- To stop it, press `^C`.

loodse

Detached Container

To start a container in detached mode, use `-d=true` or just `-d` option.

By design, containers exit when the root process used to run the container exits.

```
$ docker run -d -t -p 80:80 loodse/demo-www
```

loodse

List Containers

List all running container

```
$ docker ps
```

The `docker ps` command only shows running containers by default. To see all containers, use the `-a` (or `--all`) flag:

```
$ docker ps -a
```

loodse

Start a Second Container

Let's start a second webserver

```
$docker run -t -p 8080:80 loodse/demo-www
```

With `docker ps` you see now two running containers

loodse

ps Flags

Try out additional useful flags

```
-n, --last    int Show the last created containers (includes all states) (default -1)
-l, --latest      Show the latest created container (includes all states)
-q, --quiet       Only display numeric IDs
-s, --size        Display total file sizes
```

loodse

Stopping Containers

The main process inside the container will receive `SIGTERM`, and after a grace period, `SIGKILL`.

```
$ docker stop 0542
```

`-t, --time int` Seconds to wait for stop before killing it (default 10)

Killing Containers

The main process inside the container will be sent `SIGKILL`, or any signal specified with option `--signal`.

```
$ docker kill d032
```

Note: The `stop` and `kill` commands both can take multiple IDs.

loodse

Lab > Run a Webapp with Docker

https://www.katacoda.com/loodse/courses/docker

loodse

Lesson #5
Interacting with Containers

loodse

Default Names

Docker generates a random name from the list of adjectives and surnames

- A mood (silly, epic, trusting...)
- The name of a famous inventor (morse, stallman, montalcini...)

Examples: `eloquent_murdock, modest_bell, nifty_newton...`

Check https://github.com/moby/moby/blob/master/pkg/namesgenerator/names-generator.go

loodse

Use a Name for the Container

- Defining a name can be a handy way to add meaning to a container.
- If you specify a name, you can use it when referencing the container within a Docker network.
- This works for both background and foreground Docker containers.

```
$ docker run -d -p 80:80 --name webserver loodse/demo-www
```

Note: If the name already exist the command will fail

loodse

Check the Logs

The `docker logs` command shows you the logs of the container

```
$ docker logs webserver
```

To continue follow the new output from the container using

```
$ docker logs webserver --follow
```

View only the Tail Number of lines to show from the end of the logs

```
$ docker logs webserver --tail 1
```

loodse

Detaching from a Container

You can detach from a container and leave it running using the `CTRL-p CTRL-q` key sequence.

```
$ docker run -it --name counter loodse/counter
```

loodse

Attaching to a Container

- Use `docker attach` to attach your terminal's standard input, output, and error (or any combination of the three) to a running container using the container's ID or name.
- This allows you to view its ongoing output or to control it interactively, as though the commands were running directly in your terminal.

```
$ docker attach counter
```

Try:

```
$ docker attach $(docker ps -lq)
```

Detaching from Non-Interactive Containers

- **Warning:** if you started the container without `-it`...
  - You can't detach with `^P^Q`
  - If you hit `^C`, the signal will be proxied to the container

Note: you can always detach by killing the Docker client

Restarting a Container

When a container is in stopped, the `start` command starts stopped containers

```
$ docker start counter
```

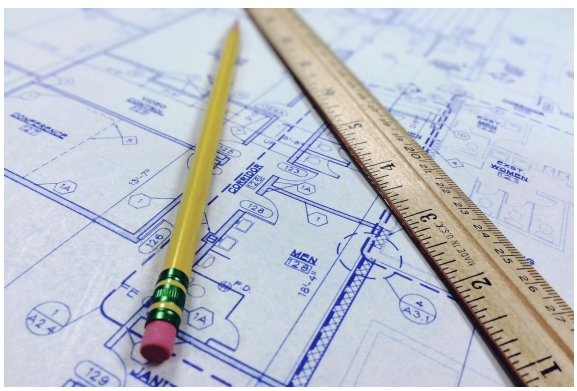
loodse

Lab > Interacting with Containers

https://www.katacoda.com/loodse/courses/docker

loodse

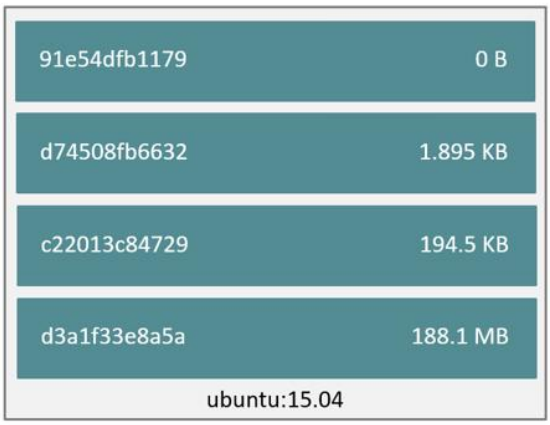
Lesson #6
Docker Images and Layers



Images and Layers

- Each Docker image references a list of read-only layers that represent filesystem differences.
- Layers are stacked on top of each other to form a base for a container's root filesystem.

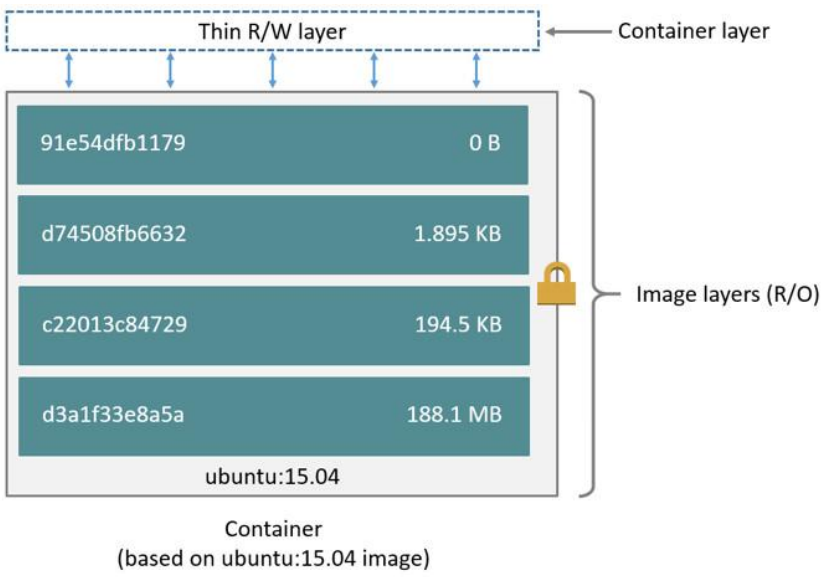
This diagram shows the Ubuntu 15.04 image comprising 4 stacked image layers:



Image

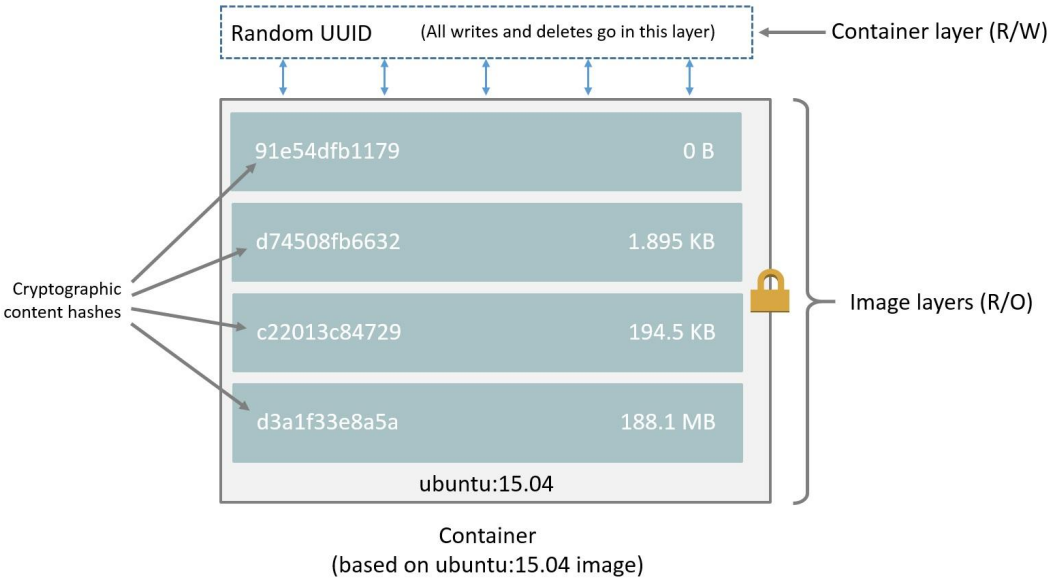
loodse

Container Layer



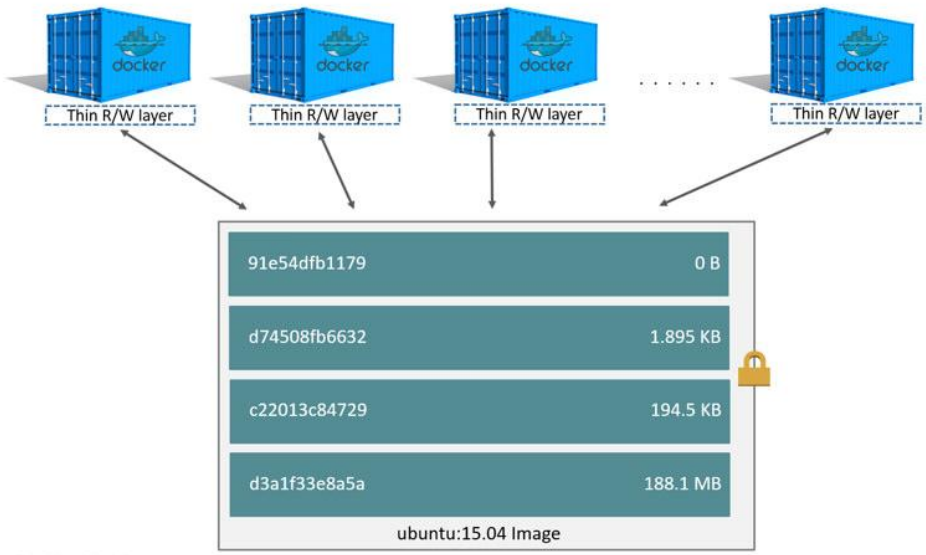
Container
(based on ubuntu:15.04 image)

Content Addressable Storage

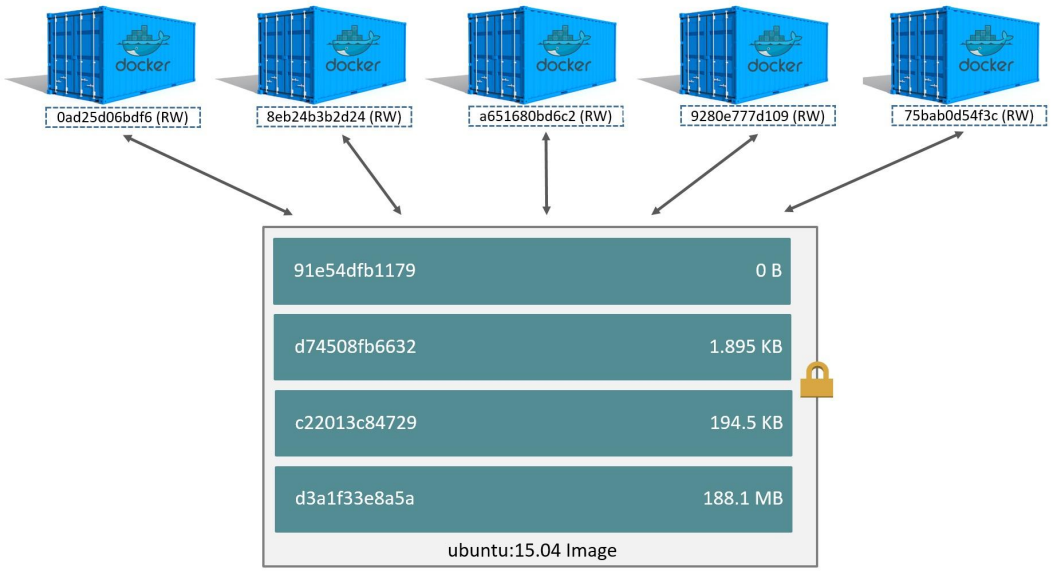


loodse

Container and Layers

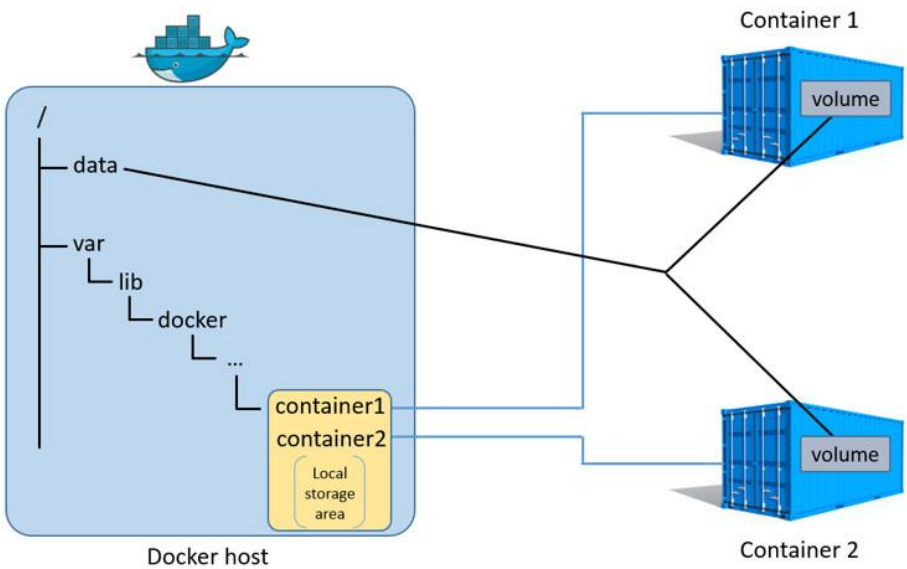


Copy-on-Write



loodse

Data Volumes and the Storage Driver



loodse

Lab > Docker Images and Layers

https://www.katacoda.com/loodse/courses/docker

loodse

Lesson #7
Building Images Interactively

Building Images Interactively

Let's build a first image with `docker commit`.

Later, we will see how to use a `Dockerfile` and `docker build`.

loodse

Building from a Base

(in our case, an `debian` image)

loodse

Start a New Container and Make Some Changes

Start an Debian container:

```
$ docker run -it debian
```

Install Apache

```
root@<yourContainerId>:#/ apt-get update && apt-get install apache2 -y
```

Type `exit` to leave the session

loodse

Inspect the Changes

Inspect changes to files or directories on a container's filesystem:

```
$ docker diff <yourContainerId>
```

This lists the changed files and directories

Commit and Run your Image

Create a new image from the container changes we created before.

```
$ docker commit <yourContainerId>
```

Run the image:

```
$ docker run -it <newImageId>
```


Tagging Images

Create a. name for your image instead of a tag id.

Create a tag by:

```
$ docker tag <newImageId> webserver
```

You can also specify the tag directly with the commit:

```
$ docker commit <containerId> webserver
```

And then run it using its tag:

```
$ docker run -it webserver
```

loodse

Images Namespaces

There are 3 namespaces:

- Official images

  e.g. `debian, busybox ...`
- User (and organizations) images

  e.g. `loodse/counter`
- Self-hosted images

  e.g. `registry.example.com:5000/my-private/image`

loodse

User Namespace

The user namespace holds images for Docker Hub users and organizations. For example:

`loodse/counter`

The Docker Hub user is:

`loodse`

The image name is:

`counter`

Self-Hosted Namespace

If you want to run your own registry the images name must contain hostname (or IP address), and optionally the port of the registry server.

For example:

```
localhost:5000/wordpress
```

- `localhost:5000` is the host and port of the registry
- `wordpress` is the name of the image

Showing Current Images

`docker images` will show all top level images, their repository and tags, and their size.

Docker images have intermediate layers that increase reusability, decrease disk usage, and speed up docker build by allowing each step to be cached. These intermediate layers are not shown by default.

```
$ docker images
```

loodse

Pulling an Image

Pull an image or a repository from a registry

- Explicitly, with `docker pull`.
- Implicitly, by executing `docker run` when the image can't be found locally

loodse

Pulling an Image

- Pulls the debain image with the tag `:jessie` from docker hub

- `$ docker pull debian:jessie`

Images and Tags

- Images can have tags
- Tags define image versions or variants
- `docker pull debian` will refer to `debian:latest`
- The `:latest` tag is usually the most up-to-date, or the current stable version

loodse

How To Store and Manage Images?

- The Docker Registry is a component of Docker's ecosystem.
- A registry is a storage and content delivery system, holding named Docker images, available in different tagged versions.
- For example, the image distribution/registry, with tags 2.0 and latest. Users interact with a registry by using docker push and pull commands.
- The Docker Hub has its own registry which, like the Hub itself, is run and managed by Docker.

When To (Not) Use Tags

Don't specify tags:

- When you're doing rapid testing and prototyping
- When you're experimenting
- When you want the latest version

Do specify tags:

- When you're recording a procedure into a script
- When you're going to production
- To make sure that the same version is used everywhere
- To ensure repeatability later

What's Next?

Generally, it is better to use `Dockerfiles` to manage your images in a documented and maintainable way.

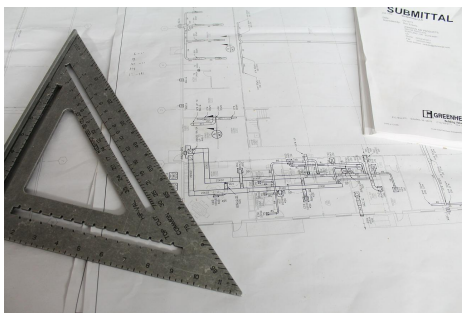
loodse

Lab > Building Images Interactively

https://www.katacoda.com/loodse/courses/docker

loodse

Lesson #8
Building Images With a Dockerfile



loodse

Dockerfile Overview

- Docker can build images automatically by reading the instructions from a Dockerfile, a text file containing all the commands, in order, needed to build a given image.
- Dockerfiles adhere to a specific format and use a specific set of instructions.

loodse

Writing a `Dockerfile`

Step 1: Create a directory for the `Dockerfile`.

```
$ mkdir myimage
```

Step 2: Create a `Dockerfile` inside this directory

```
$ cd myimage
$ vim Dockerfile
```

loodse

Type This Into Our Dockerfile...

```
FROM debian:jessie
RUN apt-get update && apt-get install apache2 -y && apt-get clean
CMD ["apache2ctl", "-DFOREGROUND"]
```

- **FROM** indicates the base image for our build
- Each **RUN** line will be executed by Docker during the build
- Our **RUN** commands must be non-interactive. (You can't provide input to Docker during the build.)
- In many cases, we will add the `-y` flag to `apt-get`.

loodse

The FROM instruction

Can specify a base image:

```
FROM <image>
Or
FROM <image>:<tag>
Or
FROM <image>@<digest>
```

The **FROM** instruction sets the Base Image for subsequent instructions. As such, a valid Dockerfile must have **FROM** as its first instruction. The image can be any valid image – it is especially easy to start by pulling an image from the Public Repositories.

loodse

The FROM instruction

- `FROM` must be the first non-comment instruction in the Dockerfile.
- `FROM` can appear multiple times within a single Dockerfile in order to use multiple stages. Build or install in one stage and `COPY --from=build` to the run-stage.
- The tag or digest values are optional. If you omit either of them, the builder assumes a latest by default. The builder returns an error if it cannot match the tag value.

loodse

The RUN instruction

`RUN` has 2 forms:

`RUN <command>` (shell form, the command is run in a shell, by default `/bin/sh -c`)

`RUN ["executable", "param1", "param2"]` (exec form)

loodse

The RUN instruction

- The **RUN** instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.

- Layering **RUN** instructions and generating commits conforms to the core concepts of Docker where commits are cheap and containers can be created from any point in an image's history, much like source control.

- The default shell for the shell form can be changed using the `SHELL` command.

loodse

The CMD instruction

The **CMD** instruction has three forms:

`CMD ["executable","param1","param2"]` (exec form, this is the preferred form)
`CMD ["param1","param2"]` (as default parameters to ENTRYPOINT)
`CMD command param1 param2` (shell form)

loodse

The CMD instruction

- There can only be one **CMD** instruction in a Dockerfile. If you list more than one **CMD** then only the last **CMD** will take effect.

- The main purpose of a **CMD** is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an **ENTRYPOINT** instruction as well.

loodse

Build It!

Now we can build our dockerfile

```
$ docker build -t webserver .
```

The `docker build` command builds an image from a Dockerfile and a context. The build's context is the files at a specified location PATH or URL. The PATH is a directory on your local filesystem. The URL is a Git repository location.

Step by Step

- The Docker daemon runs the instructions in the Dockerfile one-by-one, committing the result of each instruction to a new image if necessary, before finally outputting the ID of your new image.
- The Docker daemon will automatically clean up the context you sent.
- Note that each instruction is run independently, and causes a new image to be created - so RUN cd /tmp will not have any effect on the next instructions.

Running the Image

Test the new image

```
$ docker run -d -p 8080:80 --name www webserver
```

loodse

Build a Webapp Image

Add your own file to the image

```
$ mkdir html
$ cd html
$ vim index.html
```

loodse

Build a Webapp Image

```
<!DOCTYPE html PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML>
   <HEAD>
      <TITLE>
           Hello World
      </TITLE>
   </HEAD>
<BODY>
   <H1>Hello World !!!</H1>
</BODY>
</HTML>
```

loodse

Enhance our Dockerfile

```
FROM debian:jessie
RUN apt-get update && apt-get install apache2 -y && apt-get clean
COPY html /var/www/html

CMD ["apache2ctl", "-DFOREGROUND"]
```

loodse

The COPY Instruction

`COPY` has two forms:

`COPY <src>... <dest>`

`COPY ["<src>",... "<dest>"]` (this form is required for paths containing whitespace)

loodse

The COPY Instruction

The `COPY` instruction copies new files or directories from `<src>` and adds them to the filesystem of the container at the path `<dest>.`

Multiple `<src>` resource may be specified but they must be relative to the source directory that is being built (the context of the build).

loodse

Build and Run it Again

Now we can build our dockerfile

- `$ docker build -t webserver .`

and run it:

- `$ docker run -d -p 8080:80 --name www webserver`

loodse

Viewing History

`history` will show the history of an image with all the layer

```
$ docker history webserver
```

loodse

The Shell Form for CMD

When used in the shell or exec formats, the **CMD** instruction sets the command to be executed when running the image.

If you use the shell form of the **CMD**, then the `<command>` will execute in `/bin/sh -c:`

```
FROM debian:jessie
RUN apt-get update && apt-get install apache2 -y && apt-get clean
COPY html /var/www/html

CMD apache2ctl -DFOREGROUND
```

loodse

Build and Test Our Image

Build it:

```
$ docker build -t webserver .
```

Run it:

```
$ docker run -t -p 80:80 webserver
```

loodse

The exec form for CMD

If you want to run your `<command>` without a shell, you must express the command as a JSON array and give the full path to the executable.

This array form is the preferred format of **CMD**. Any additional parameters must be individually expressed as strings in the `array: c:`

```
FROM debian:jessie
RUN apt-get update && apt-get install apache2 -y && apt-get clean
COPY html /var/www/html

CMD ["apache2ctl", "-DFOREGROUND"]
```

The exec vs shell for CMD

- If `CMD` is used to provide default arguments for the `ENTRYPOINT` instruction, both the `CMD` and `ENTRYPOINT` instructions should be specified with the JSON array format.

- The exec form is parsed as a JSON array, which means that you must use double-quotes (") around words not single-quotes (').

- Unlike the shell form, the exec form does not invoke a command shell. This means that normal shell processing does not happen. For example, `CMD [ "echo", "$HOME" ]` will not do variable substitution on `$HOME`.
- If you want shell processing then either use the shell form or execute a shell directly, for example: `CMD [ "sh", "-c", "echo $HOME" ].` When using the exec form and executing a shell directly, as in the case for the shell form, it is the shell that is doing the environment variable expansion, not docker.

loodse

Overriding CMD

If the user specifies arguments to `docker run` then they will override the default specified in `CMD.`

You can replaced the `apache2ctl` with e.g. bash

```
$ docker run -it webserver bash
```

loodse

The ENTRYPOINT Instruction

`ENTRYPOINT` has two forms:

`ENTRYPOINT ["executable", "param1", "param2"]` (*exec* form, preferred)

`ENTRYPOINT command param1 param2` (*shell* form)

loodse

The CMD instruction

- Command line arguments to `docker run <image>` will be appended after all elements in an `exec` form `ENTRYPOINT`, and will override all elements specified using `CMD`.
- This allows arguments to be passed to the entry point, i.e., `docker run <image> -d` will pass the `-d` argument to the entry point.
- You can override the `ENTRYPOINT` instruction using the `docker run --entrypoint` flag.
- The shell form prevents any `CMD` or `run` command line arguments from being used, but has the disadvantage that your `ENTRYPOINT` will be started as a subcommand of `/bin/sh -c`, which does not pass signals.
- This means that the executable will not be the container's PID 1 - and will not receive Unix signals - so your executable will not receive a `SIGTERM` from `docker stop <container>.`

loodse

Exec **form** ENTRYPOINT

You can use the `exec` form of `ENTRYPOINT` to set fairly stable default commands and arguments and then use either form of `CMD` to set additional defaults that are more likely to be changed.

```
FROM debian:jessie
RUN apt-get update && apt-get install apache2 -y && apt-get clean
COPY html /var/www/html

ENTRYPOINT ["apache2ctl", "-e"]
```

loodse

Build and Test Our Image

Build it:

```
$ docker build -t webserver .
```

Run it:

`$ docker run -t -p 80:80 webserver debug` (try warn)

loodse

Shell form ENTRYPOINT

- You can specify a plain string for the `ENTRYPOINT` and it will execute in `/bin/sh -c`.
- This form will use shell processing to substitute shell environment variables, and will ignore any CMD or docker run command line arguments.
- To ensure that `docker stop` will signal any long running `ENTRYPOINT` executable correctly, you need to remember to start it with `exec`

```
FROM debian:jessie
RUN apt-get update && apt-get install apache2 -y && apt-get clean
COPY html /var/www/html

ENTRYPOINT exec apache2ctl -e debug
```

loodse

Build and Test Our Image

Build it:

```
$ docker build -t webserver .
```

Run it:

```
$ docker run -t -p 80:80 webserver
```

loodse

Understand how CMD, ENTRYPOINT interact

Both `CMD` and `ENTRYPOINT` instructions define what command gets executed when running a container. There are few rules that describe their co-operation.

- `Dockerfile` should specify at least one of `CMD` or `ENTRYPOINT` commands.
- `ENTRYPOINT` should be defined when using the container as an executable.
- `CMD` should be used as a way of defining default arguments for an `ENTRYPOINT` command or for executing an ad-hoc command in a container.
- `CMD` will be overridden when running the container with alternative arguments.

loodse

Using CMD, ENTRYPOINT together

- `ENTRYPOINT` defines the base command for the container
- `CMD` defines the default parameters for the command.

```
FROM debian:jessie
RUN apt-get update && apt-get install apache2 -y && apt-get clean
COPY html /var/www/html

ENTRYPOINT ["apache2ctl", "-e"]
CMD ["debug"]
```

loodse

Build and Test Our Image

Build it:

```
$ docker build -t webserver .
```

Run it:

```
$ docker run -t -p 80:80 webserver
```

and

```
$ docker run -t -p 80:80 webserver warn
```

loodse

Overriding ENTRYPOINT

You can replace the `ENTRYPOINT` with `--entrypoint` during the run command

```
$ docker run -it --entrypoint /bin/bash -p 80:80 webserver
```

Lab > Building Images with Dockerfile

https://www.katacoda.com/loodse/courses/docker

loodse

Lab > Docker Build ignore Files

https://www.katacoda.com/loodse/courses/docker

loodse

Lesson #9
Copying files during the build

Objectives

So far, we have installed things in our container images by downloading packages.

We can also copy files from the build context to the container that we are building.

Remember: the build context is the directory containing the Dockerfile.

In this chapter, we will learn a new Dockerfile keyword: `COPY.`

loodse

Build some C code

We want to build a container that compiles a basic "Hello world" program in C. Here is the program, `hello.c`:

```
int main () {
  puts("Hello, world!");
  return 0;
}
```

Let's create a new directory, and put this file in there.

Then we will write the Dockerfile.

loodse

The Dockerfile

On Debian, the package `build-essential` will get us a compiler.

When installing it, don't forget to specify the -y flag, otherwise the build will fail (since the build cannot be interactive).

Then we will use `COPY` to place the source file into the container.

```
FROM debian
RUN apt-get update
RUN apt-get install -y build-essential
COPY hello.c /
RUN make hello
CMD /hello
```

Create this Dockerfile.

loodse

Testing our C program

- Create `hello.c` and `Dockerfile` in the same directory.
- Run `docker build -t hello .` in this directory.
- Run `docker run hello`, you should see `Hello, world!`.

COPY and the build cache

- Run the build again.
- Now, modify `hello.c` and run the build again.
- Docker can cache steps involving `COPY`.
- Those steps will not be executed again if the files haven't been changed

Details

- You can `COPY` whole directories recursively.
- Older Dockerfiles also have the `ADD` instruction. It is similar but can automatically extract archives.
- If we really wanted to compile C code in a compiler, we would:
    - Place it in a different directory, with the `WORKDIR` instruction.
    - Even better, use the `gcc` official image.

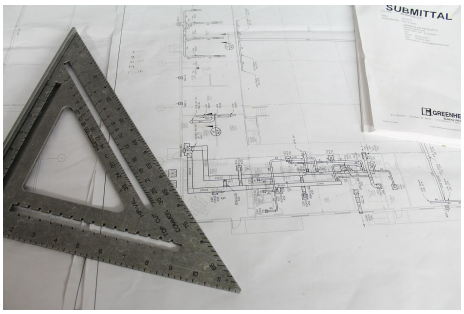
loodse

Freestyle Lab > Playground

https://www.katacoda.com/loodse/courses/docker

loodse

Lesson #9
Advanced Dockerfiles



loodse

Advanced Dockerfiles

Objectives:

We have used some basic Dockerfiles to demonstrate how Docker builds container images. Now we will see:

- Additional keywords and their syntax for use in Dockerfiles.
- Best practices to write Dockerfiles.

loodse

`Dockerfile` usage summary

- `Dockerfile` instructions are executed in order.
- Each instruction in the Dockerfile creates a new image layer.
- Build instructions are cached. The instruction will only be executed on subsequent builds if the layer no longer exists or if the instruction has changed.
- The `FROM` instruction MUST be the first non-comment instruction. Subsequent FROM instructions can be used for multi-stage builds
- Lines prefixed # are interpreted as comments.
- While you can specify more than one `CMD` or `ENTRYPOINT` instruction in a `Dockerfile` only the last one will be used.

loodse

The FROM instruction

- Defines the base image to layer on top of.
- Must be the first instruction line in the `Dockerfile`

loodse

The FROM instruction

Can specify a base image:

```
FROM debian
```

A tagged image:

```
FROM debian:jessie
```

A user image:

```
FROM loodse/counter
```

Or a self-hosted image:

```
FROM localhost:5000/monkey
```

loodse

More about FROM

- The **FROM** instruction can be specified multiple times to build staged images.

```
FROM debian:jessie
. . .
FROM fedora:23
. . .
```

- Also know as Multi-stage builds
- Each **FROM** instruction marks the beginning of the build in a new image.
- Tags only are applied to the last image in the build chain.
- If the build fails, existing tags are left unchanged.
- A version tag can be used with the name of the image. E.g.: `debian:jessie`.

loodse

A use case for multiple FROM lines

- Integrate CI and unit tests in the build system

```
FROM <baseimage>
RUN <retrieve dependencies>
COPY <repository>
RUN <build repository>
RUN < retrieve test dependencies>
COPY <test data>
RUN <unit tests>
FROM <baseimage>
RUN <install dependencies>
COPY <tested code>
RUN <build code>
CMD, EXPOSE ...
```

- The build stops at the first instruction failure
- If **RUN <unit tests>** fails no image is produced from the build
- If all instructions succeed a clean image (without test libraries and data) is produced

loodse

A use case for multiple FROM lines

- Files can be copied from one stage to another by `COPY --from=0`

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

loodse

The RUN instruction

The **RUN** instruction can be specified in JSON or string syntax.

With string syntax the instruction is wrapped with `/bin/sh -c:`

```
RUN apt-get install figlet
```

With JSON syntax there is no shell string expansion, allowing execution in images that don't contain `/bin/sh`:

```
RUN [ "apt-get", “install" , “figlet“ ]
```

loodse

More about the RUN instruction

`RUN` does the following:

- Executes a command.
- Commits changes made to the filesystem.
- Useful for installing libraries, packages and making changes to the underlying filesystem.

`RUN` does NOT do the following:

- Retain state of any processes.
- Automatically execute daemons.

Starting something automatically when the container runs is the domain of `CMD` and/or `ENTRYPOINT`.

loodse

Collapsing layers

You can execute multiple commands in a single build step

```
RUN apt-get update && apt-get install -y curl && apt-get clean
```

You can also break a command onto multiple lines for readability:

```
RUN apt-get update \
 && apt-get install -y curl \
 && apt-get clean
```

loodse

The EXPOSE instruction

The `EXPOSE` instruction tells Docker what ports are to be published in this image

```
EXPOSE 8080
EXPOSE 80 443
EXPOSE 53/tcp 53/udp
```

- Ports are private by default.
- The `Dockerfile` doesn't determine if a port is publicly available when the container is run.
- When you `docker run -p <port>` ..., that port is exposed even if it was not declared with `EXPOSE`.
- When you `docker run -P` ... (without a port number), all ports declared with `EXPOSE` become public.
- A public port is reachable from other containers and from outside the host.
- A private port is not reachable from other containers or outside the host.

loodse

The COPY instruction

The `COPY` instruction adds files from the build context into the image

```
COPY ./bin
```

This will add the contents of the build context (the directory passed as an argument to `docker build`) to the directory `/bin` in the container.

Note: you can only reference files and directories inside the build context. Absolute paths are taken as being anchored to the build context, so the two following lines are equivalent:

```
COPY . /bin
COPY / /bin
```

Attempts to use .. to get out of the build context will be blocked by Docker causing the build to fail.

If a build was allowed to break out of the build context a `Dockerfile` could succeed on one host, but fail on another.

loodse

ADD

ADD is very similar to COPY, but has a few additional features.
ADD can download remote files:

```
ADD http://www.example.com/app.jar /opt/
```

would download the `app.jar` file and place it in the `/opt` directory.
ADD will automatically extract zip and tar archives:

```
ADD ./resources.zip /var/www/htdocs/resources/
```

would extract `resources.zip into /var/www/htdocs/resources`.
However, ADD does not unpack remote archives.

loodse

ADD , COPY, and the build cache

- With most Dockerfile instructions Docker checks if the line in the Dockerfile has changed.
- For ADD and COPY, Docker checks if the files specified have been changed as well.
- When ADD refers to a remote file it must download it before it can check for changes. It does not check ETags or If-Modified-Since headers.

VOLUME

The `VOLUME` instruction tells Docker that a directory should be treated as a volume

```
VOLUME /var/lib/mysql
```

Access in volumes bypasses the copy-on-write layer, allowing native I/O performance for files in those directories.

Volumes can be attached to multiple containers, allowing "porting" of data from one container to another, e.g. to upgrade a database.

It is possible to start containers in "read-only" mode. The container filesystem is read-only, but volumes will still have read/write access.

The `WORKDIR` instruction

The `WORKDIR` instruction sets the working directory for subsequent instructions.

`WORKDIR` also affects relative paths in `CMD` and `ENTRYPOINT`, as it sets the directory used when starting the container.

```
WORKDIR /bin
```

You can specify `WORKDIR` multiple times to change the directory for further operations

The ENV instruction

The `ENV` instruction defines environment variables set in any container launched from the image.

```
ENV APP_PORT 8080
```

This will result in an environment variable being created in any containers created from this image of

```
APP_PORT=8080
```

You can also specify environment variables when you use `docker run`

```
$ docker run -e APP_PORT=8080 -e APP_HOST=www.example.com ...
```

Environment variables specified at runtime override those set in the build

The USER instruction

The `USER` instruction sets the UID or username used when running the container

`USER` can be specified several times in the `Dockerfile` to switch between user contexts (i.e. root) for specific operations

loodse

The CMD instruction

The CMD instruction sets the default command to be run when the container created.

```
CMD [ ”redis-server", ”--sentinel", ”--loglevel=verbose" ]
```

Means we don't need to specify `redis-server --sentinel --loglevel=verbose` when running the container.

Instead of:

```
$ docker run <dockerhubUsername>/redis-sentinel redis-server --sentinel
--loglevel=verbose
```

We can just do:

```
$ docker run <dockerhubUsername>/redis-sentinel
```

loodse

More on CMD

Just like **RUN**, the **CMD** instruction offers two syntax options. The string syntax will wrap the command in a shell

```
CMD redis-server --sentinel --loglevel=verbose
```

The second executes directly, without shell processing:

```
CMD [ ”redis-server", ”--sentinel", ”--loglevel=verbose" ]
```

loodse

Overriding the CMD instruction

You can override `CMD` when you start a container.

```
docker run -it <dockerhubUsername>/redis bash
```

Will run `bash` instead of `redis-server --sentinel --loglevel=verbose`

loodse

The ENTRYPOINT instruction

The `ENTRYPOINT` instruction similar to the `CMD` instruction, but arguments to the `RUN` command are appended to the `ENTRYPOINT` instead of overriding it.

`ENTRYPOINT` requires the use of JSON syntax

```
ENTRYPOINT [ "/bin/echo" ]
```

When we run:

```
docker run loodse/echo monkey
```

Instead of trying to run `monkey`, the container will run `/bin/echo monkey`

loodse

Overriding the ENTRYPOINT instruction

You can also override the command specified in the `ENTRYPOINT` instruction

```
$ docker run -it loodse/echo

$ docker run -it --entrypoint bash loodse/echo
```

loodse

How CMD and ENTRYPOINT interact

The `CMD` and `ENTRYPOINT` instructions can be combined to provide clear directives.

```
ENTRYPOINT [ "redis-server" ]
CMD [”--sentinel", ”--loglevel=verbose" ]
```

The `ENTRYPOINT` specifies the command to be executed and `CMD` provides the arguments. At runtime we can then optionally override `CMD` with our own values.

```
$ docker run -d <dockerhubUsername>/redis --port=6379
```

The resulting command executed in the container will be `redis-server --port=6379`

loodse

Advanced Dockerfile instructions

- `ONBUILD` defines instructions executed when the resulting image is used in a `FROM` instruction.
- `LABEL` adds arbitrary metadata to the image.
- `ARG` defines build-time variables (optional or mandatory).
- `STOPSIGNAL` sets the signal for `docker stop` (`TERM` by default).
- `HEALTHCHECK` defines a command assessing the status of the container.
- `SHELL` sets the default program to use for string-syntax `RUN`, `CMD`, etc.

loodse

The `ONBUILD` instruction

The `ONBUILD` instruction is a trigger. It sets instructions that will be executed when another image is built from the current image.

This is useful for building images which will be used as base-images for other builds.

```
ONBUILD COPY . /src
```

- You can't chain `ONBUILD` instructions with `ONBUILD`.
- `ONBUILD` can't be used to trigger `FROM` and `MAINTAINER` instructions.

loodse

Building an efficient `Dockerfile`

- Each line in a `Dockerfile` creates a new layer.
- Arrange your `Dockerfile` to take advantage the cache.
- Combine logical sets of commands by using `&&` and `\` to escape linefeeds.
- `COPY` dependency lists (`package.json`,`pom.xml`, etc.) in their own instruction to prevent reinstalling unchanged dependencies on every build.

loodse

Example "bad" `Dockerfile`

The dependencies are reinstalled every time, because the build system does not know if `pom.xml` has been updated.

```
FROM openjdk
COPY . /src/
WORKDIR /src
RUN mvn build
EXPOSE 5000
CMD [”java", ”-jar”, “app.jar”]
```

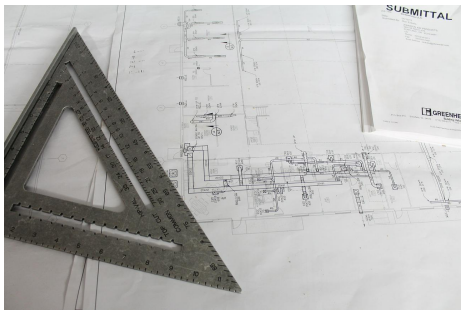
Fixed Dockerfile

Adding the dependencies as a separate step means that Docker can cache more efficiently and only install them when `pom.xml` changes.

```
FROM openjdk
COPY ./pom.xml /src/pom.xml
WORKDIR /src
RUN mvn build
EXPOSE 5000
CMD [”java", ”-jar”, “app.jar”]
```

loodse

Lesson #10
Docker Hub



Uploading our images to the Docker Hub

We have built our first images, and we could share those images through the Docker Hub.

In order to do this the steps would be:

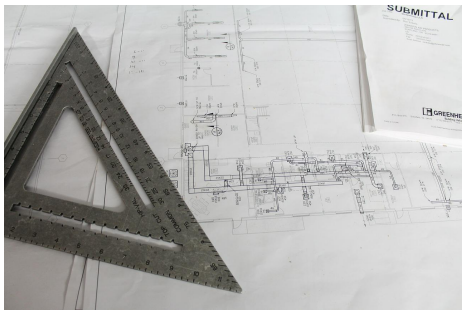
- Create an account on the Docker Hub
- Tag our image accordingly (i.e. `username/imagename`)
- `docker push username/imagename`

Anybody can now `docker run username/imagename` from any Docker host.

Docker Hub offers private image hosting as well, for a fee.

loodse

Lesson #11
Naming and Inspecting



loodse

Naming and Inspecting

Naming allows us to:

- Easily reference a container.
- Ensure unicity of a specific container.

We will also see the `inspect` command, which provides extensive information on the container

loodse

Naming our containers

So far, we have always referenced containers only with their ID or with a prefix, but containers can also be referenced by its name.

Every container is assigned a name, either user provided or a default name that is randomly generated

If a container is named `dev-redis`, I can do:

```
$ docker start dev-redis
$ docker logs dev-redis
etc.
```

loodse

Inspecting a container

The `docker inspect` command will output a JSON map with all the details about the container.

```
$ docker inspect <containerID>
[{
"AppArmorProfile": "",
"Args": [],
"Config": {
     "AttachStderr": true,
     "AttachStdin": false,
     "AttachStdout": true,
     "Cmd": [
     "bash"
     ],
     "CpuShares": 0,
...
```

There are several ways to parse that information.

loodse

Parsing JSON with the Shell

Use `jq` to parse the output

```
$ docker inspect <containerID> | jq .
```

Using `--format`

You can provide a format string, which will be parsed by Go's text/template package.

```
$ docker inspect --format '{{ json .Created }}' <containerID>
```

- The generic syntax is to wrap the expression with double curly braces.
- The expression starts with a dot representing the JSON object.
- Then each field or member can be accessed in dotted notation syntax.
- The optional `json` keyword defines that we want valid JSON output

loodse

Lab > Naming and Inspecting

https://www.katacoda.com/loodse/courses/docker

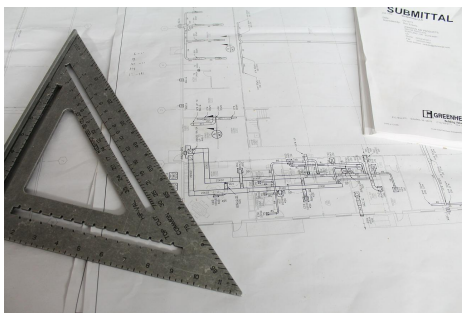
loodse

Lab > Managing Log Files

https://www.katacoda.com/loodse/courses/docker

loodse

Lesson #12
Networking Fundamentals



loodse

Networking Fundamentals

Learning Objectives

- Run network services in a container
- Change a basic container network
- Identify a container's IP address

loodse

Basic Web Server

Execute the `nginx` image which contains a simple web server

```
$ docker run -d -P nginx
```

- The image is downloaded from Docker Hub
- `-d` runs the image in the background
- `-P` makes the service accessible from other computers (EXPOSE)

loodse

Web server port

You run `docker ps` to find the web server port

```
$ docker ps
```

- The Nginx container uses ports 80 and 443 which are mapped to ports on the Docker host

loodse

Connecting to Nginx by Browser

Go to the IP address of your Docker host and the port that is mapped to container port 80. (`docker ps`)

loodse

Connecting to Nginx with curl

It is also possible to use `curl localhost` to connect to Nginx directly from the Docker host.

```
$ curl localhost:<mapped port>
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

loodse

Reasons for mapping ports

- IPv4 addresses are costly
- Containers are not assigned public IPv4 addresses.
- Containers are assigned private (and not public) IPv4 addresses
- Ports expose services in the containers
- Docker maps ports to avoid collisions.

loodse

Locating the Nginx Port in a Script

Docker client provides a helpful command:

```
$ docker port <containerID> 80
```

loodse

Allocate Port Numbers Manually

There is the possibility for manual allocation:

```
$ docker run -d -p 80:80 nginx
$ docker run -d -p 6443:443 nginx
$ docker run -d -p 8080:80 -p 8443:443 nginx
```

- We run 2 NGINX web servers, one being exposed on port 80, the other on port 6443.
- The third web server is exposed on ports 8080 and 8443.

Convention: `host-port:container-port.`

loodse

Integrating Containers into Our Infrastructure

There are different options:

- Start a container with `-P.` Take the exposed port assigned by Docker. Use it in your configuration.
- Specify port numbers in your configuration and set the ports manually when starting the container.
- Use a network plugin to connect containers to e.g. VLANs, tunnels…

loodse

Locating the IP Address

You have `docker inspect` to determine your container's IP Address:

```
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' <yourContainerID>
172.17.0.3
```

loodse

Test Connectivity

We can use the `ping` command to test connectivity via the IP:

```
$ ping <ipAddress>
```

loodse

Networking Drivers

You can run a container on top of several network drivers such as:

- `bridge` (default)
- `none`
- `host`
- `container`

Select the driver with `docker run --net` ....

The Bridge Driver

- The container gets a virtual `eth0` interface by default, which is provided by a `veth` pair and connected to the Docker bridge (Default name `docker0`; configurable with `--bridge`.)
- Addresses are allocated on a private, internal subnet. (`172.17.0.0/16` by default; configurable with `--bip`.)
- Iptables `MASQUERADE` rule for outbound traffic
- Iptables `DNAT` rule for inbound traffic
- Containers can have their own routes, and iptables rules…

The Null Driver

- Start container with `docker run --net none` …
- Only sees loopback interface `No eth0`.
- No possibility to send and receive network traffic.
- Helpful for isolated or untrusted workloads

The Host Driver

- Start container with `docker run --net host` ...
- It sees the network interfaces of the host and can access them
- It can bind any address any any port
- Network traffic doesn't have to go through NAT, bridge, or veth.
- Performance = native!

Use cases:

- Apps that are performance sensitive e.g. gaming or streaming
- For peer discovery (Raft, Serf...)

The Container Driver

- Start: `docker run --net container:id` ...
- Re-uses the network stack of another container and shares interfaces,, routes, iptables rules...with it
- They are able to communicate over their `lo` interface

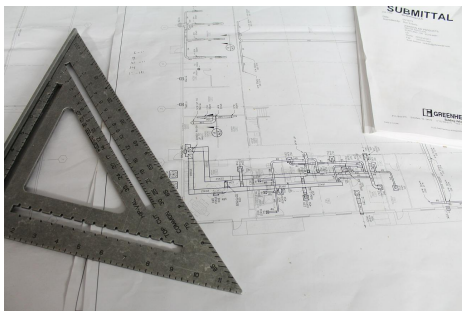
loodse

Lab > Naming and Inspecting

https://www.katacoda.com/loodse/courses/docker

loodse

Lesson #13
Local Development Workflow



loodse

Local Development Workflow

Learning Objectives

- Sharing of code between container and host
- Basic local development workflows

loodse

Why using containers for local development?

With containers, we have a consistent environment for development.

That means you can finally forget about:

"Works on my machine"
"Not the same version"
"Missing dependency"

loodse

Our "namer" application

The code is available on https://github.com/jpetazzo/namer .
The image loodse/namer is automatically built by the Docker Hub.

Let's run it with:

```
$ docker run -dP jpetazzo/namer
```

Check the port number with `docker ps` and open the application.

loodse

Let's look at the code

Download the source code of the app:

```
$ git clone https://github.com/jpetazzo/namer
$ cd namer
$ ls -1
company_name_generator.rb
config.ru
docker-compose.yml
Dockerfile
Gemfile
```

loodse

Where's my code?

According to the Dockerfile, the code is copied into `/src` :

```
FROM ruby
MAINTAINER Education Team at Docker <education@docker.com>
COPY . /src  WORKDIR /src
RUN bundler install
CMD ["rackup", "--host", "0.0.0.0"]  EXPOSE 9292
```

We want to make changes *inside the container* without rebuilding it each time. For that, we will use a *volume*.

loodse

Our first volume

We will tell Docker to map the current directory to `/src` in the container.

```
$ docker run -d -v $(pwd):/src -p 80:9292 jpetazzo/namer
```

The `-d` flag indicates that the container should run in detached mode (in the background).
The `-v` flag provides volume mounting inside containers.
The `-p` flag maps port 9292 inside the container to port 80 on the host.
`jpetazzo/namer` is the name of the image we will run.
We don't need to give a command to run because the Dockerfile already specifies `rackup`.

loodse

Mounting Volumes

You can mount a directory from your host into your container with the `-v` flag:

```
[host-path]:[container-path]:[rw|ro]
```

If [host-path] or [container-path] do not exist yet, they will be created. Check the write status of the volume with `ro` and `rw` (`rw` by default in case `rw` or `ro` are not specified)

loodse

Testing the Development Container

Let's test it:

```
$ docker ps
```

loodse

Viewing the Application

Now let's browse to our web application on:

```
http://<yourHostIP>:80
```

We can see our company naming application.

loodse

Making a change to our application

Our customer really doesn't like the color of our text. Let's change it.

```
$ vi company_name_generator.rb
```

And change

```
color: royalblue;
```

To:

```
color: red;
```

loodse

Refreshing our application

Now let's refresh our browser:

```
http://<yourHostIP>:80
```

We can see the updated color of our company naming application.

loodse

Improving the workflow with Compose

You can also start the container with the following command:

```
$ docker-compose up -d
```

This works thanks to the Compose file, docker-compose.yml:

```
www:
    build: .
    volumes:
    - .:/src
    ports:
    - 80:9292
```

Reasons for Using Compose?

Specifying all those "docker run" parameters is tedious.

And error-prone.

We can "encode" those parameters in a "Compose file."

When you see a `docker-compose.yml` file, you know that you can use `docker-compose up`.

Compose can also deal with complex, multi-container apps. (More on this later.)

Workflow explained

We can see a simple workflow:

1. Build an image containing our development environment. (Rails, Django...)
2. Start a container from that image.
   Use the `-v` flag to mount source code inside the container.
3. Edit source code outside the containers, using regular tools. (vim, emacs, textmate...)
4. Test application.
   (Some frameworks pick up changes automatically.
   Others require you to Ctrl-C + restart after each modification.)
5. Repeat last two steps until satisfied.
6. When done, commit+push source code changes. (You *are* using version control, right?)

loodse

Debugging inside the container

In 1.3, Docker introduced a feature called `docker exec`.

It allows users to run a new process in a container which is already running.

If sometimes you find yourself wishing you could SSH into a container: you can use `docker exec` instead.

You can get a shell prompt inside an existing container this way, or run an arbitrary process for automation.

loodse

docker exec example

You can run ruby commands in the area the app is running and more!

```
$ docker exec -it <yourContainerId> bash
root@5ca27cf74c2e:/opt/namer# irb
irb(main):001:0> [0, 1, 2, 3, 4].map {|x| x ** 2}.compact
=> [0, 1, 4, 9, 16]
irb(main):002:0> exit
```

Stopping the container

Now that we're done let's stop our container.

```
$ docker stop <yourContainerID>
```

And remove it.

```
$ docker rm <yourContainerID>
```


Section summary

We've learned how to:

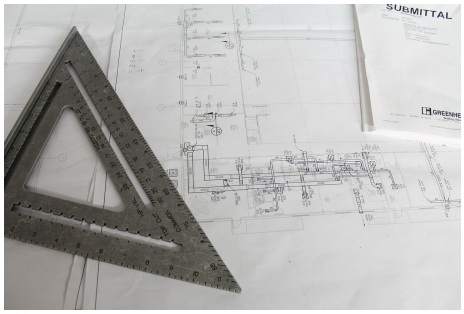
- Share code between container and host.
- Set our working directory.
- Use a simple local development workflow.

Freestyle Lab > Playground

https://www.katacoda.com/loodse/courses/docker

loodse

Lesson #14
Docker Don’ts



Docker "don't"-s

- Don't store data inside of containers - they're ephemeral
- Don't split your app - keep the whole app in one container
- Don't create large images - the smaller the better
- Don't use only the latest tag - add version tags, use them for deployments
- Don't use docker commit. Seriously, no. Just don't do it. Use Dockerfiles.
- Don't run more then one process in a single container

loodse

Build 12 factor apps!

https://12factor.net

loodse

Lesson #15
Advanced Networking

Objectives

We will learn about the CNM (Container Network Model). At the end of this lesson, you will be able to:

- Create a private network for a group of containers.
- Use container naming to connect services together.
- Dynamically connect and disconnect containers to networks.
- Set the IP address of a container.

We will also explain the principle of overlay networks and network plugins.

loodse

The Container Network Model

The CNM was introduced in Engine 1.9.0 (November 2015).

The CNM adds the notion of a network, and a new top-level command to manipulate and see those networks: `docker network`.

```
$ docker network ls
```

loodse

What's in a network?

- Conceptually, a network is a virtual switch.
- It can be local (to a single Engine) or global (across multiple hosts).
- A network has an IP subnet associated to it.
- A network is managed by a driver.
- A network can have a custom IPAM (IP allocator).
- Containers with explicit names are discoverable via DNS.
- All the drivers that we have seen before are available.
- A new multi-host driver, overlay, is available out of the box.
- More drivers can be provided by plugins (OVS, VLAN...)

loodse

Creating a network

Let's create a network called `dev`.

```
$ docker network create dev
```

The network is now visible with the network `ls` command:

```
$ docker network ls
```

loodse

Placing containers on a network

We will create a *named* container on this network.

It will be reachable with its name, search.

```
$ docker run -d --name search --net dev elasticsearch
```

loodse

Communication between containers

Now, create another container on this network.

```
$ docker run -ti --net dev alpine sh
```

From this new container, we can resolve and ping the other one, using its assigned name:

```
/ # ping search
```

loodse

Resolving container addresses

In Docker Engine 1.9, name resolution is implemented with `/etc/hosts`, and updating it each time containers are added/removed.

```
[root@0eaaadfa45ef   /]# cat /etc/hosts
```

In Docker Engine 1.10, this has been replaced by a dynamic resolver. (This avoids race conditions when updating `/etc/hosts`.)

loodse

Connecting multiple containers together

Let's try to run an application that requires two containers.

The first container is a web server.

The other one is a redis data store.

We will place them both on the `dev` network created before.

loodse

Running the web server

The application is provided by the container image `loodse/trainingwheels`.
We don't know much about it so we will try to run it and see what happens!

Start the container, exposing all its ports:

```
$ docker run --net dev -d -P loodse/trainingwheels
```

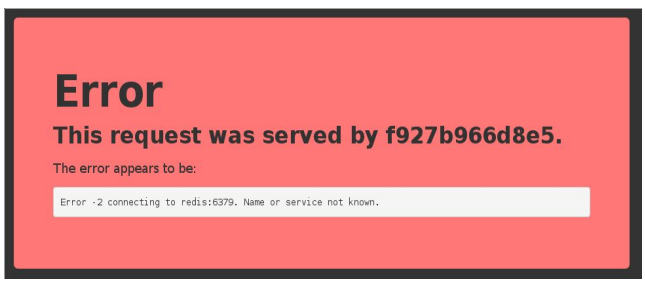
Check the port that has been allocated to it:

```
$ docker ps -l
```

loodse

Test the web server

If we connect to the application now, we will see an error page:



This is because the Redis service is not running.
This container tries to resolve the name redis.

Note: we're not using a FQDN or an IP address here; just redis.

loodse

Start the data store

We need to start a Redis container.
That container must be on the same network as the web server.
It must have the right name (redis) so the application can find it.

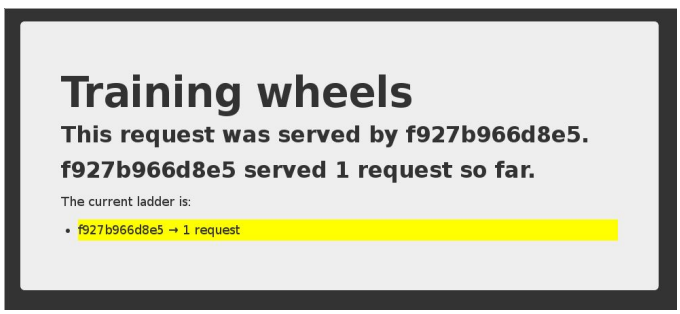
Start the container:

```
$ docker run --net dev --name redis -d redis
```

loodse

Test the web server again

If we connect to the application now, we should see that the app is working correctly:



When the app tries to resolve `redis`, instead of getting a DNS error, it gets the IP address of our Redis container.

loodse

A few words on scope

What if we want to run multiple copies of our application?

Since names are unique, there can be only one container named `redis` at a time.

We can specify `--net-alias` to define network-scoped aliases, independently of the container name.

Let's remove the `redis` container:

```
$ docker rm -f redis
```

And create one that doesn't block the `redis` name:

```
$ docker run --net dev --net-alias redis -d redis
```

Check that the app still works (but the counter is back to 1, since we wiped out the old Redis container).

loodse

Names are `local` to each network

Let's try to ping our search container from another container, when that other container is *not* on the dev. network.

```
$ docker run --rm alpine ping search
```

Names can be resolved only when containers are on the same network.

Containers can contact each other only when they are on the same network (you can try to ping using the IP address to verify).

loodse

Network aliases

We would like to have another network, `prod`, with its own `search` container. But there can be only one container named `search`!

We will use *network aliases*.

A container can have multiple network aliases.

Network aliases are *local* to a given network (only exist in this network).

Multiple containers can have the same network alias (even on the same network). In Docker Engine 1.11, resolving a network alias yields the IP addresses of all containers holding this alias.

loodse

Creating containers on another network

Create the `prod` network.

```
$ docker create network prod
```

We can now create multiple containers with the `search` alias on the new `prod` network.

```
$ docker run -d --name prod-es-1 --net-alias search --net prod elasticsearch
```

```
$ docker run -d --name prod-es-2 --net-alias search --net prod elasticsearch
```

loodse

Resolving network aliases

Let's try DNS resolution first, using the **nslookup** tool that ships with the **alpine** image.

```
$ docker run --net prod --rm alpine nslookup search
```

(You can ignore the **can't resolve '(null)'** errors.)

loodse

Connecting to aliased containers

Each ElasticSearch instance has a name (generated when it is started). This name can be seen when we issue a simple HTTP request on the ElasticSearch API endpoint.

Try the following command a few times:

```
$ docker run --rm --net dev centos curl -s
```

Then try it a few times by replacing `--net dev` with `--net prod`:

```
$ docker run --rm --net prod centos curl -
```

Good to know ...

Docker will not create network names and aliases on the default `bridge` network.

Therefore, if you want to use those features, you have to create a custom network first.

Network aliases are *not* unique: you can give multiple containers the same alias *on the same network*.

- In Engine 1.10: one container will be selected and only its IP address will be returned when resolving the network alias.
- In Engine 1.11: when resolving the network alias, the DNS reply includes the IP addresses of all containers with this network alias. This allows crude load balancing across multiple containers (but is not a substitute for a real load balancer).
- In Engine 1.12: enabling *Swarm Mode* gives access to clustering features, including an advanced load balancer using Linux IPVS.

Creation of networks and network aliases is generally automated with tools like Compose (covered in a few chapters).

loodse

A few words about round robin DNS

Don't rely exclusively on round robin DNS to achieve load balancing. Many factors can affect DNS resolution, and you might see:

all traffic going to a single instance;
traffic being split (unevenly) between some instances;
different behavior depending on your application language;
different behavior depending on your base distro;
different behavior depending on other factors (sic).

It's OK to use DNS to discover available endpoints, but remember that you have to re- resolve every now and then to discover new endpoints.

loodse

Custom networks

When creating a network, extra options can be provided.

`--internal` disables outbound traffic (the network won't have a default gateway).

`--gateway` indicates which address to use for the gateway (when utbound traffic is allowed).

`--subnet` (in CIDR notation) indicates the subnet to use.

`--ip-range` (in CIDR notation) indicates the subnet to allocate from.

`--aux-address` allows to specify a list of reserved addresses (which won't be allocated to containers).

loodse

Setting containers' IP address

It is possible to set a container's address with --ip.
The IP address has to be within the subnet used for the container.

A full example would look like this.

```
$ docker network create --subnet 10.66.0.0/16 pubnet
$ docker run --net pubnet --ip 10.66.66.66 -d nginx
```

Note: don't hard code container IP addresses in your code!
I repeat: don't hard code container IP addresses in your code!

loodse

Overlay networks

The features we've seen so far only work when all containers are on a single host.

If containers span multiple hosts, we need an *overlay* network to connect them together.

Docker ships with a default network plugin, overlay, implementing an overlay network leveraging VXLAN.

Other plugins (Weave, Calico...) can provide overlay networks as well.

Once you have an overlay network, *all the features that we've used in this chapter work identically.*

loodse

Multi-host networking (overlay)

Out of the scope for this intro-level workshop! Very short instructions:

enable Swarm Mode (`docker swarm init` then `docker swarm join` on other nodes)

```
docker network create mynet --driver overlay
docker service create --network mynet myimage
```

Multi-host networking (plugins)

Out of the scope for this intro-level workshop! General idea:

install the plugin (they often ship within containers)

run the plugin (if it's in a container, it will often require extra parameters; don't just `docker run` it blindly!)

some plugins require configuration or activation (creating a special file that tells Docker "use the plugin whose control socket is at the following location")

you can then `docker network create --driver pluginname`

How `links` work

Links are created *between two containers*

Links are created *from the client to the server*

Links associate an arbitrary name to an existing container

Links exist *only in the context of the client*

Freestyle Lab > Playground

https://www.katacoda.com/loodse/courses/docker

loodse

Lesson #16
Working with Volumes

Objectives

At the end of this lesson, you will be able to:

- Create containers holding volumes
- Share volumes across containers
- Share a host directory with one or many containers.

loodse

Working with Volumes

Docker volumes can be used to achieve many things, including:

Bypassing the copy-on-write system to obtain native disk `I/O` performance.

Bypassing copy-on-write to leave some files out of `docker commit`.

Sharing a directory between multiple containers.

Sharing a directory between the host and a container.

Sharing a *single file* between the host and a container.

loodse

Volumes are special directories in a container

Volumes can be declared in two different ways.

Within a `Dockerfile`, with a `VOLUME` instruction.

```
VOLUME /uploads
```

On the command-line, with the `-v` flag for `docker run`.

```
$ docker run -d -v /uploads myapp
```

In both cases, `/uploads` (inside the container) will be a volume.

loodse

Volumes bypass the copy-on-write system

Volumes act as passthroughs to the host filesystem.

The `I/O` performance on a volume is exactly the same as `I/O` performance on the Docker host.

When you `docker commit`, the content of volumes is not brought into the resulting image.

If a `RUN` instruction in a `Dockerfile` changes the content of a volume, those changes are not recorded neither.

If a container is started with the `--read-o`nly flag, the volume will still be writable (unless the volume is a read-only volume).

loodse

Volumes can be shared across containers

You can start a container with *exactly the same volumes* as another one.
The new container will have the same volumes, in the same directories.
They will contain exactly the same thing, and remain in sync.
Under the hood, they are actually the same directories on the host anyway.
This is done using the `--volumes-from` flag for `docker run`.

```
$ docker run -it --name alpha -v /var/log debian bash
```

In another terminal, let's start another container with the same volume.

```
$ docker run --volumes-from alpha debian cat /var/log/now
```

loodse

Volumes exist independently of containers

If a container is stopped, its volumes still exist and are available.
Since Docker 1.9, we can see all existing volumes and manipulate them:

```
$ docker volume ls
```

Some of those volume names were explicit (pgdata-prod, pgdata-dev).

The others (the hex IDs) were generated automatically by Docker.

loodse

Data containers (before Engine 1.9)

A data container is a container created for the sole purpose of referencing one (or many) volumes.
It is typically created with a no-op command:

```
$ docker run --name files -v /var/www busybox true
$ docker run --name logs -v /var/log busybox true
```

- We created two data containers.
- They are using the `busybox` image, a tiny image.
- We used the command `true`, possibly the simplest command in the world!
- We named each container to reference them easily later.

Using data containers

Data containers are used by other containers thanks to `--volumes-from`.

Consider the following (fictitious) example, using the previously created volumes:

```
$ docker run -d --volumes-from files --volumes-from logs webserver
$ docker run -d --volumes-from files ftpserver
$ docker run -d --volumes-from logs lumberjack
```

The first container runs a webserver, serving content from `/var/www` and logging to `/var/log`.

The second container runs a FTP server, allowing to upload content to the same `/var/www path`.

The third container collects the logs, and sends them to logstash, a log storage and analysis system, using the lumberjack protocol.

Named volumes (since Engine 1.9)

- We can now create and manipulate volumes as first-class concepts.
- Volumes can be created without a container, then used in multiple containers.

Let's create a volume directly.

```
$ docker volume create --name=website
```

Volumes are not anchored to a specific path.

loodse

Using our named volumes

- Volumes are used with the -v option.
- When a host path does not contain a /, it is considered to be a volume name.

Let's start a web server using the two previous volumes.

```
$ docker run -d -p 8888:80 \
-v website:/usr/share/nginx/html \
-v logs:/var/log/nginx \
nginx
```

Check that it's running correctly:

```
$ curl localhost:8888
```

loodse

Using a volume in another container

- We will make changes to the volume from another container.
- In this example, we will run a text editor in the other container, but this could be a FTP server, a WebDAV server, a Git receiver...

Let's start another container using the website volume.

```
$ docker run -v website:/website -w /website -ti alpine vi index.html
```

Make changes, save, and exit.

Then run `curl localhost:8888` again to see your changes.

Managing volumes explicitly

In some cases, you want a specific directory on the host to be mapped inside the container:

- You want to manage storage and snapshots yourself. (With LVM, or a SAN, or ZFS, or anything else!)
- You have a separate disk with better performance (SSD) or resiliency (EBS) than the system disk, and you want to put important data on that disk.
- You want to share your source directory between your host (where the source gets edited) and the container (where it is compiled or executed).

Wait, we already met the last use-case in our example development workflow! Nice.

```
$ docker run -d -v /path/on/the/host:/path/in/container image ...
```

loodse

Sharing a directory between the host and a container

The previous example would become something like this:

```
$ mkdir -p /mnt/files /mnt/logs
$ docker run -d -v /mnt/files:/var/www -v /mnt/logs:/var/log webserver
$ docker run -d -v /mnt/files:/home/ftp ftpserver
$ docker run -d -v /mnt/logs:/var/log lumberjack
```

Note that the paths must be absolute.
Those volumes can also be shared with `--volumes-from`

loodse

Migrating data with `--volumes-from`

The `--volumes-from` option tells Docker to re-use all the volumes of an existing container.

- Scenario: migrating from Redis 2.8 to Redis 3.0.
- We have a container (`myredis`) running Redis 2.8.
- Stop the `myredis` container.
- Start a new container, using the Redis 3.0 image, and the `--volumes-from` option.
- The new container will inherit the data of the old one.
- Newer containers can use `--volumes-from` too.

loodse

Data migration in practice

Let's create a Redis container.

```
$ docker run -d --name redis28 redis:2.8
```

Connect to the Redis container and set some data.

```
$ docker run -ti --link redis28:redis alpine telnet redis 6379
```

Issue the following commands:

```
SET counter 42
INFO server
SAVE
QUIT
```

loodse

Upgrading Redis

Stop the Redis container.

```
$ docker stop redis28
```

Start the new Redis container.

```
$ docker run -d --name redis30 --volumes-from redis28 redis:3.0
```

loodse

Testing the new Redis

Connect to the Redis container and see our data.

```
docker run -ti --link redis30:redis alpine telnet redis 6379
```

Issue a few commands.

```
GET counter
INFO server
QUIT
```

loodse

What happens when you remove containers with volumes?

- With Engine versions prior 1.9, volumes would be orphaned when the last container referencing them is destroyed.
- Orphaned volumes are not deleted, but you cannot access them. (Unless you do some serious archaeology in `/var/lib/docker`.)
- Since Engine 1.9, orphaned volumes can be listed with `docker volume ls` and mounted to containers with `-v`.

Ultimately, you are the one responsible for logging, monitoring, and backup of your volumes.

loodse

Checking volumes defined by an image

Wondering if an image has volumes? Just use docker inspect:

```
$ # docker inspect loodse/datavol
[{
"config": {
. . .
"Volumes": {
"/var/webapp": {}
},
. . .
```

Checking volumes used by a container

To look which paths are actually volumes, and to what they are bound, use `docker inspect` (again):

```
$ docker inspect <yourContainerID>
```

- We can see that our volume is present on the file system of the Docker host

Check container status

It can be tedious to check the status of your containers with docker ps, especially when running multiple apps at the same time.

Compose makes it easier; with `docker-compose ps` you will see only the status of the containers of the current stack:

```
$ docker-compose ps
```

loodse

Cleaning up

If you have started your application in the background with Compose and want to stop it easily, you can use the `kill` command:

```
$ docker-compose kill
```

Likewise, `docker-compose rm` will let you remove containers (after confirmation):

```
$ docker-compose rm
```

Alternatively, `docker-compose down` will stop and remove containers.

```
$ docker-compose down
```

loodse

Special handling of volumes

Compose is smart. If your container uses volumes, when you restart your application, Compose will create a new container, but carefully re-use the volumes it was using previously.

This makes it easy to upgrade a stateful service, by pulling its new image and just restarting your stack with Compose.

Lab > Persisting Data Using Volumes

https://www.katacoda.com/loodse/courses/docker

loodse

Tipps & Tricks

loodse

Use Container Images as Build Artefacts

1. Build your app from Dockerfiles
2. Store images in a registry and keep them as long as you want
3. Test images in CI, QA, integration...
4. Run the same images in production
5. Not working as intended? Easily rollback to previous image

Note: Images contain all the libraries, dependencies, etc. needed to run the app

loodse

Decouple "plumbing" from App Logic

1. Write your code to connect to **named services** ("db", "api"...)
2. Use **Docker Compose** to start your stack
3. **Docker** sets up per-container DNS resolver for those names
4. You can scale, add load balancers, replication ... without needing to change your code

Security

CIS Guides:

see the attached files in the shared folder

- CIS Docker Benchmark

CIS Benchmark Tools

- docker/docker-bench-security: Docker CIS

Course Summary

During this class, we:

- Installed Docker.
- Launched our first container.
- Learned about images.
- Got an understanding about how to manage connectivity and data in Docker containers.
- Learned how to integrate Docker into your daily workflow.

Questions & Next Steps

- Docker homepage - http://www.docker.com/
- Docker Hub - https://hub.docker.com
- Docker blog - http://blog.docker.com/
- Docker documentation - http://docs.docker.com/
- Docker Getting Started Guide - http://www.docker.com/gettingstarted/
- Docker code on GitHub - https://github.com/docker/docker
- Docker mailing list - https://groups.google.com/forum/#!forum/docker-user
- Docker on IRC: irc.freenode.net and channels #docker and #docker-dev
- Docker on Twitter - http://twitter.com/docker
- Get Docker help on Stack Overflow - http://stackoverflow.com/search?q=docker