Lab 3. Docker Orchestration

—

Total Virtualization course

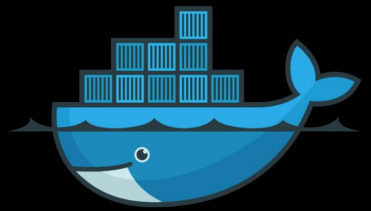
Instructor: Anna Melekhova Assistant: Vadim Reutskiy

Docker orchestration is not a magic ;)

Agenda

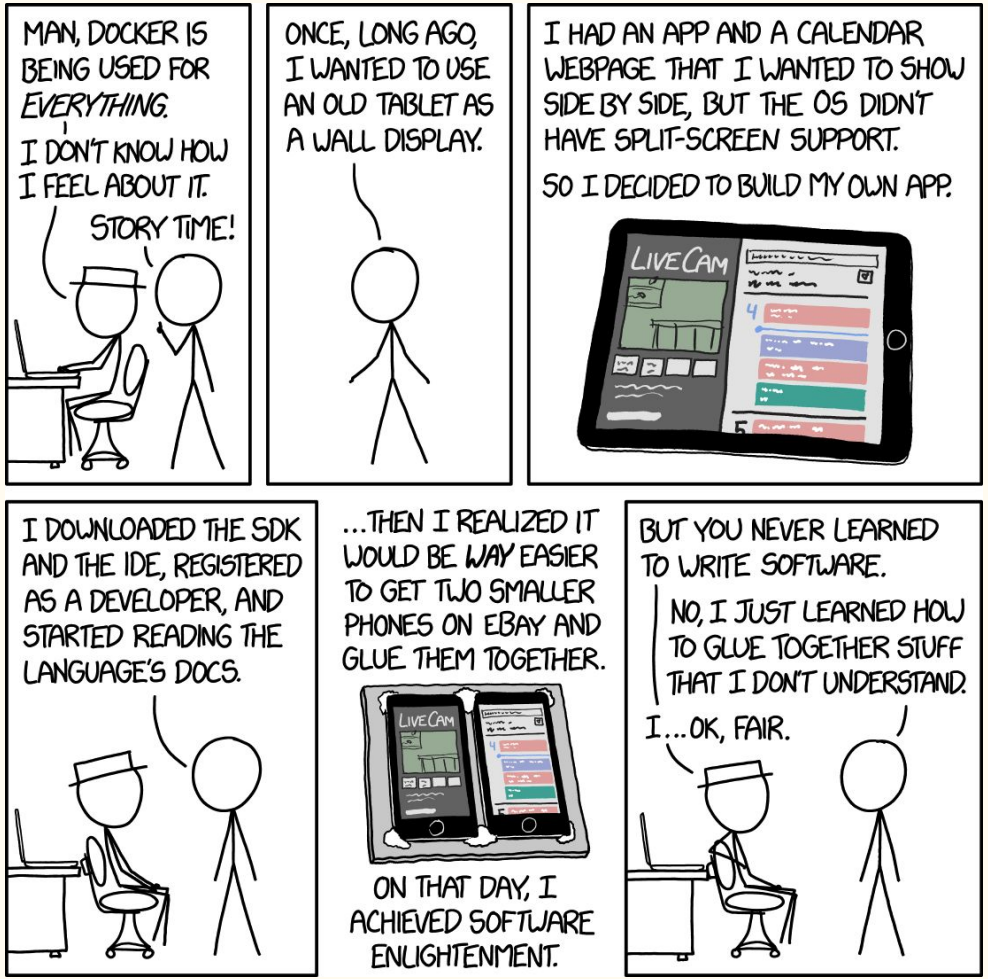
- Docker Basics
- How it works
- Use Cases
- Docker Drawbacks
- Building Docker Container
- Docker Compose
- Composition configuration
- Docker Swarm
- Swarm Configuration

But first, these slides:
https://goo.gl/etrDmD



Docker

Docker



Source: https://xkcd.com/1988/

Docker

Docker is just a tool for adding virtualization layer for your application.

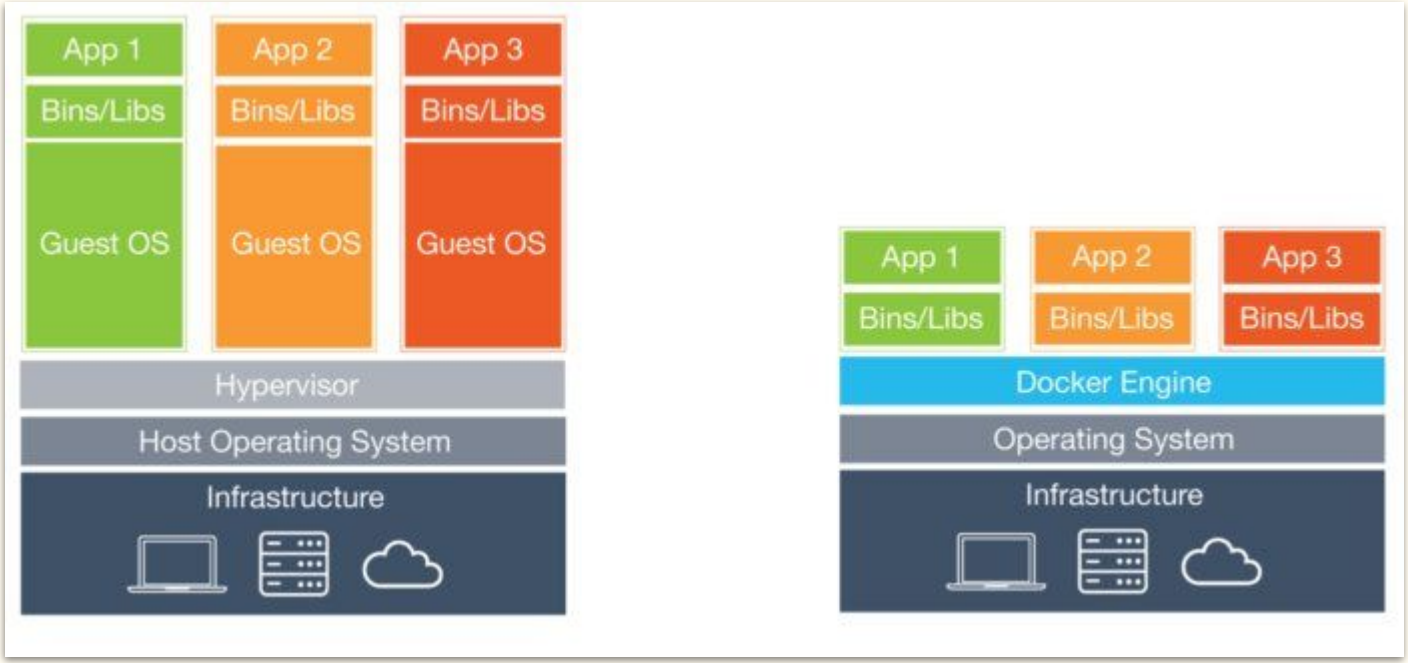
Plus unprecedented toolkit and ecosystem of services for supporting all possible needs of application delivery and execution.

Docker killer-features

- Almost no overhead for “virtualization”
- Fast to distribute using repository or image files
- The end of environment-specific problems – build and run on any Docker-compatible machine
- Easy control of containers from console or code through Docker Engine API
- Versioning and caching for images
- Running Linux containers on Windows server

Docker: How it works?

Docker VS Virtual Machine



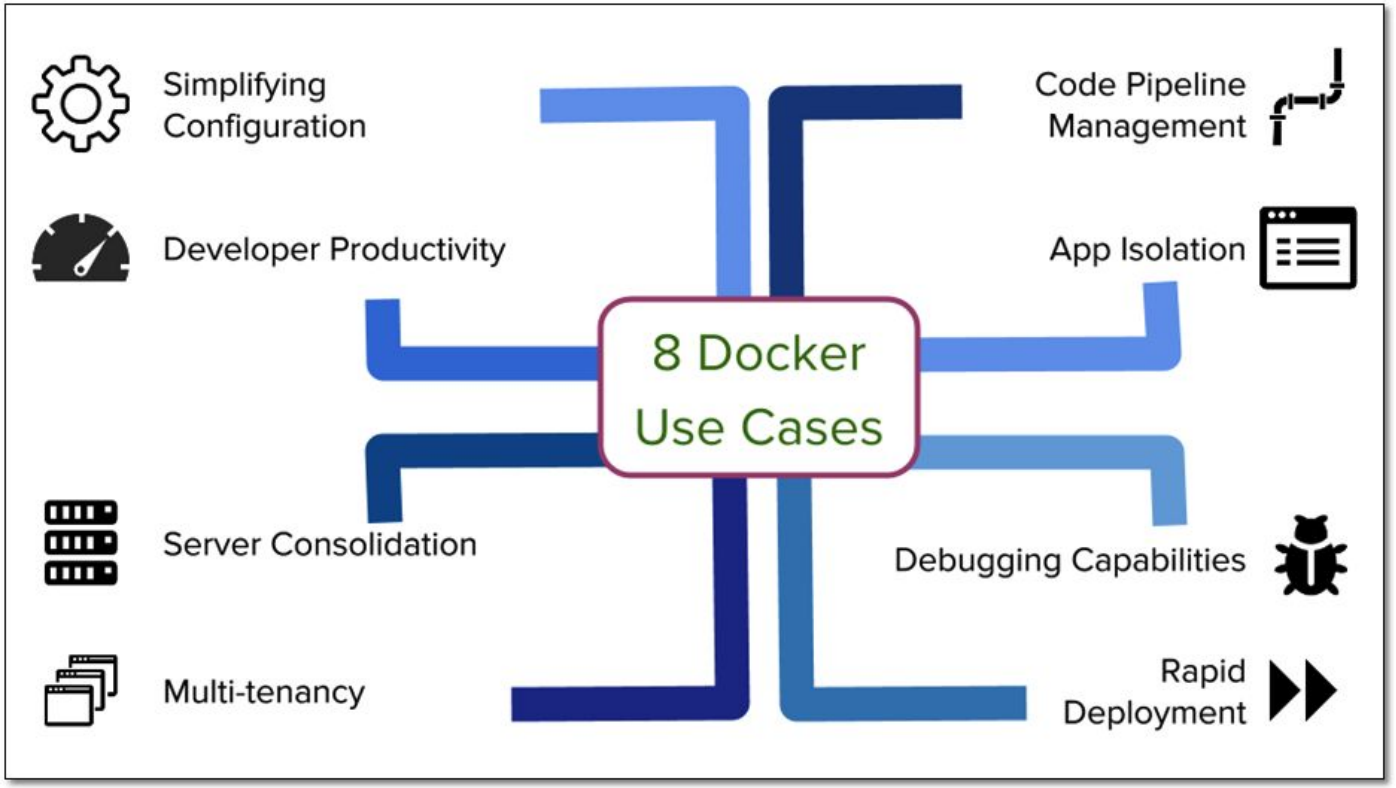
Virtual Machine

Docker

Source: https://www.infoworld.com/article/3204171/linux/what-is-docker-linux-containers-explained.html

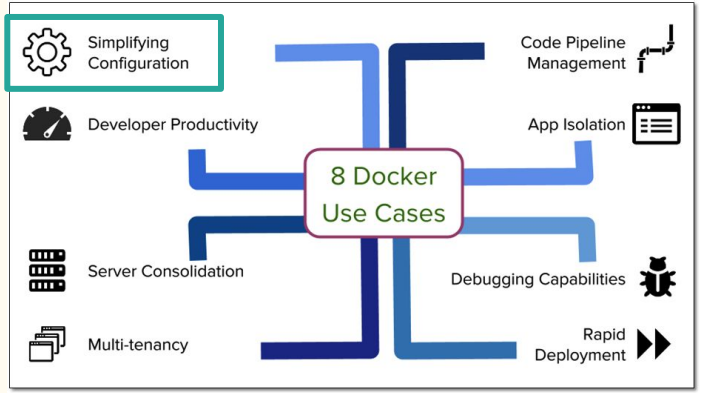
Docker: Use Cases

Many of them



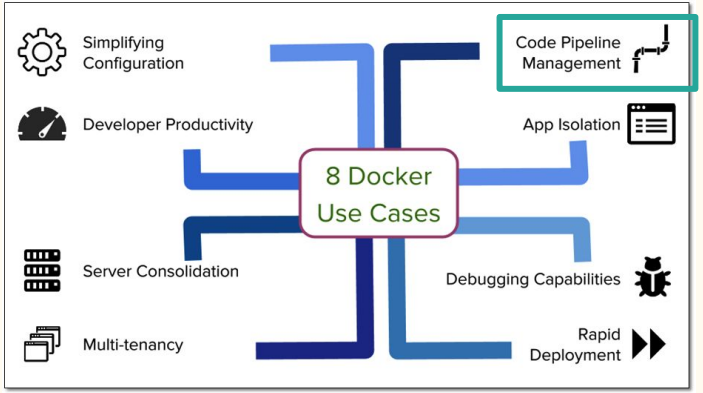
Source: http://blog.flux7.com/blogs/docker/8-ways-to-use-docker-in-the-real-world\

Simplifying configuration



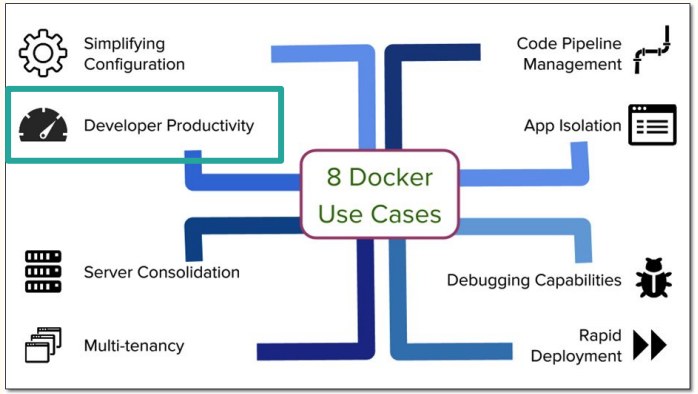
The obvious one - you can easily control the configuration of your application environment right from the code, and then use it on any Docker-friendly platform (many of them).

Code pipelining



Because of usage of the same configuration in every code delivery step, there is no more pain with travelling of code from developers' machines to testing, staging and production.

Developer Productivity



During product development every developer needs two things:

1. Environment should be as close as possible to production
2. Development tools should be fast in aim not to wait for eternity

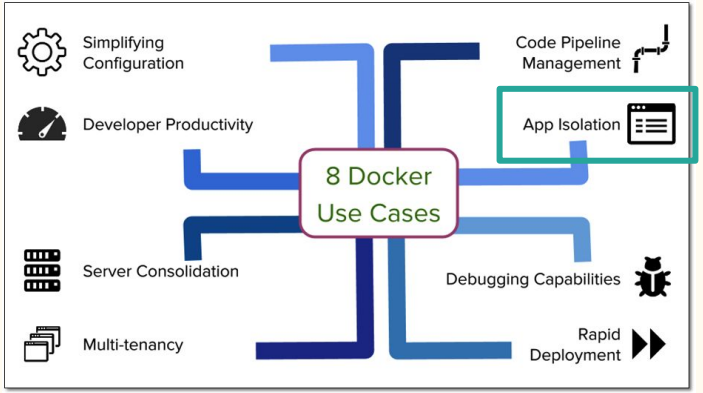
Dockers cover (or helps a lot to cover) both of them.

Application isolation

Dependency hell - check.

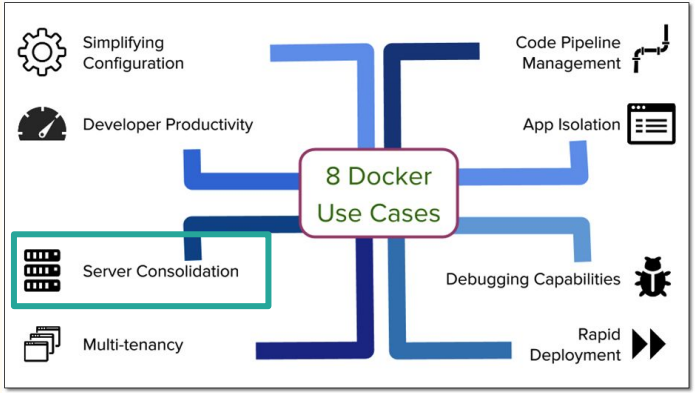
Application decoupling - check.

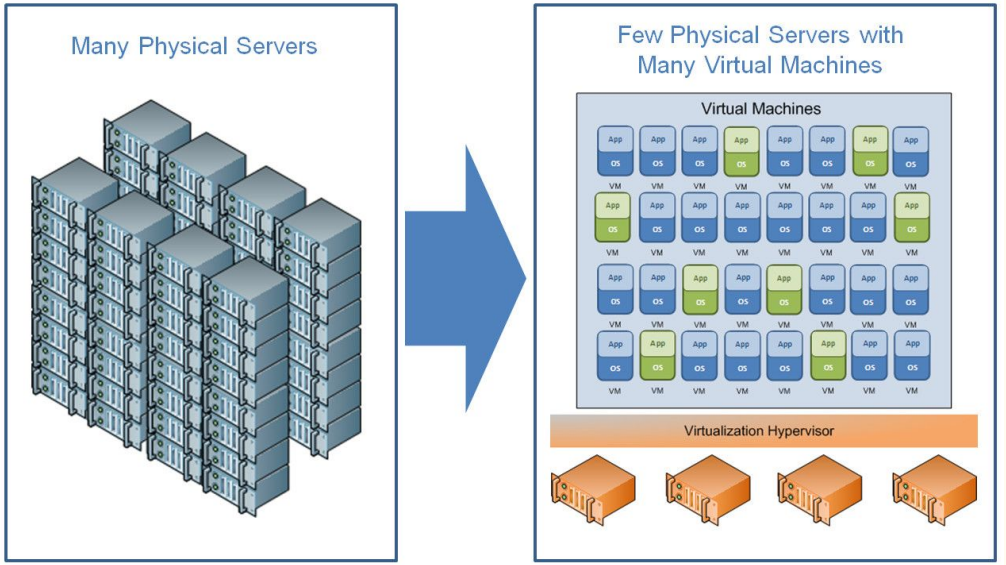
Building microservice architecture - check.



Server consolidation

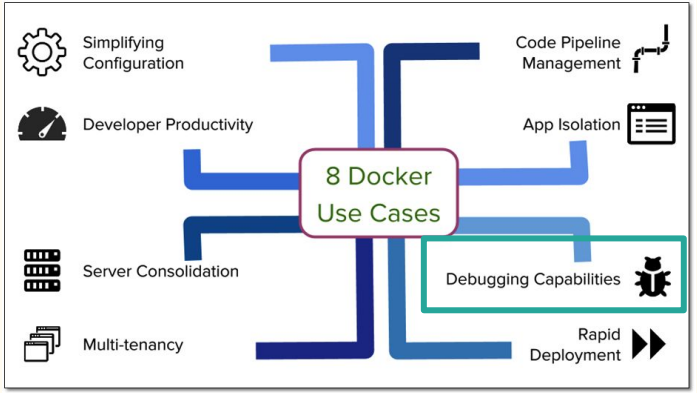
Using containers (with their low cost of virtualization) and microservices (with their possibility to make smaller modules of application) companies can utilize servers more rationally because of higher density of services.





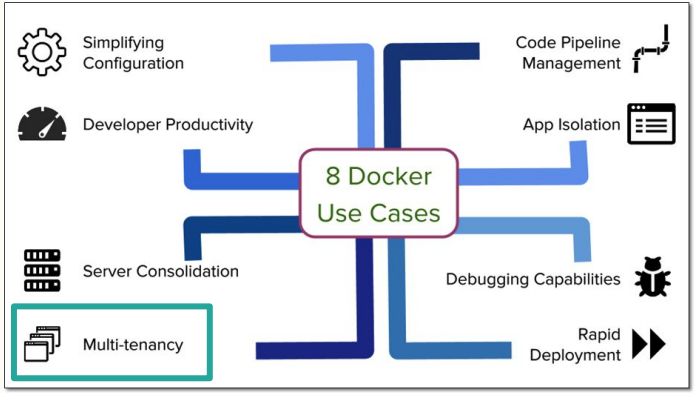
Source: http://g3it.me/solution/virtualization

Debugging



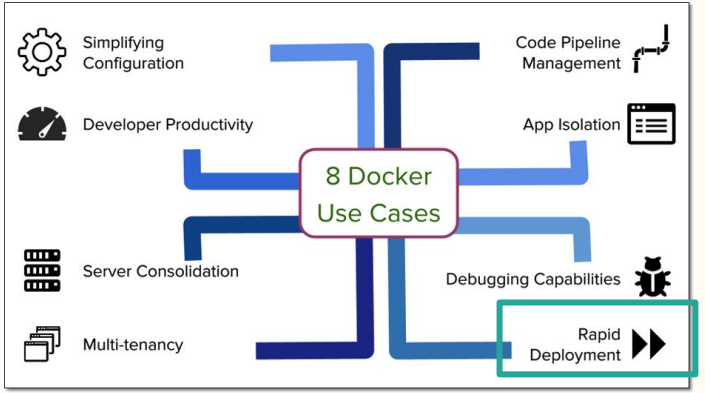
Possibility of keeping track for changes of containers (versioning) and comparing them between each other increases speed of locating source of issue.

Multi-tenancy



Each tier of multi-tenant application can be put into isolated container, which helps with development (one tier to single team), maintenance (in case of problem only single part of system will fail) and reuse.

Rapid deployment



Several seconds – time needed to boot up a container.

Vertical scaling could not be faster.

Changing of data center configuration could not be less painful.

Docker: Drawbacks

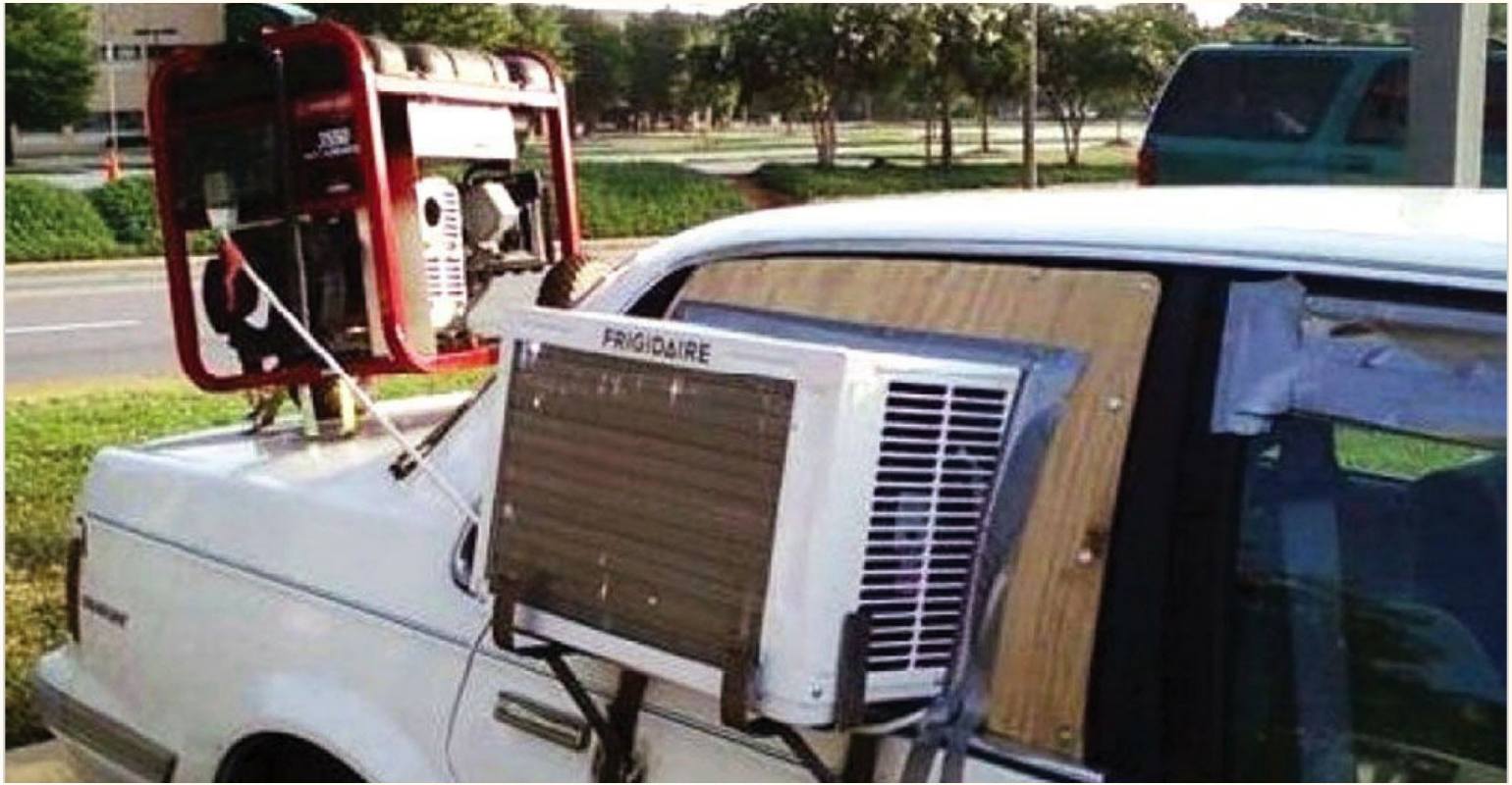
Increased system complexity

(and potential sources of problems)

To migrate your project to microservice architecture and containerisation you need to perform much work and became familiar with several new technologies.

Each of them can became a source of problems in future.

Microservice architecture can be bad



Overhead for environment

If you have N microservices, which works on JVM each consuming M bytes of RAM, you need N*M bytes of RAM for running whole system.

Moreover, CPU consumption is not ideal, too.

It is better than VMs, but still worse than bare metal solutions.

Security

Docker daemon runs as root by default. “Container breakout” can happen eventually.

Container base images, downloaded from repository, can bring danger inside.

Security issues of services, which fixed in upstream, could stay way longer in your images without noticing.

Managing sensitive information need much attention.

Source and more info: https://sysdig.com/blog/7-docker-security-vulnerabilities/

Managing big amount of containers

By itself it can became a problem for your new ‘DevOps’ role.

However, there are several solutions: Kubernetes, Apache Mesos, etc.

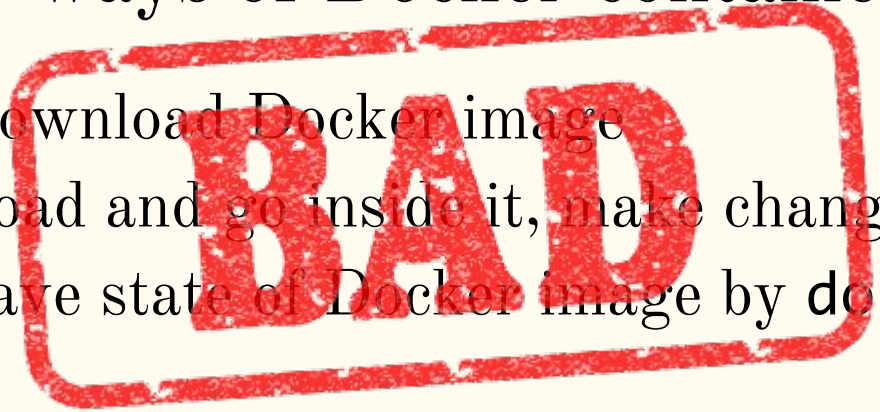
So, plus one technology, with time to master, potential problems and own drawbacks.

Your meaning

Does Docker and containerization worth it?

Building Docker container

Two ways of Docker container creation



1. Download Docker image
2. Load and go inside it, make changes to environment
3. Save state of Docker image by `docker commit`

OR

1. Write Dockerfile
2. Build image from Dockerfile
3. (Optionally) Put it into Docker Hub

Why bad? https://hackernoon.com/to-commit-or-not-to-commit-5ab72f9a466e

Language of Dockerfile

FROM *{base_image_name_from_repository}*

WORKDIR *{path_to_workdir_for_further_commands}*

ADD *{source_file_or_url}* *{destination_path_in_image}*

RUN *{command_to_run_for_environment_configuration}*

CMD *{default_command_to_run_at_image_startup}*

More information: https://docs.docker.com/engine/reference/builder/

Dockerfile example

FROM node:latest

ADD /home/user/. /app

WORKDIR /app

RUN npm install

EXPOSE 80

CMD ["npm","start"]

Dockerfile example

FROM node:latest *# download image with preinstalled Node*

ADD /home/user/. /app *# copy project into /app folder*

WORKDIR /app *# set current directory to /app*

RUN npm install *# install NPM project dependencies*

EXPOSE 80 *# listen for 80 port*

CMD ["npm","start"] *# “docker run” will execute this*

Build & Run

```
> docker build -t my-image:tag .

> docker images

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my-image            tag                 f15829dc4b0b        15 seconds ago      643 MB

> docker run my-image -d

-d means “Run image in detached mode (in background)”

> docker ps
```

More information: https://docs.docker.com/engine/reference/run/

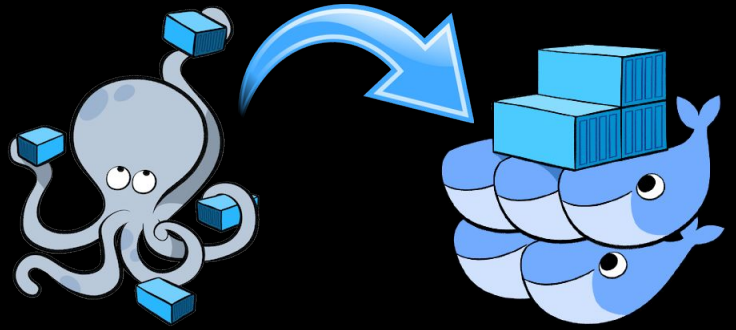
Dockerfile best practices

- Create independent and flexible container images (do not hardcode arguments)
- Follow Single Responsibility principle - do not include several applications inside single container
- Be specific in imports, keep them as narrow as possible
- Pass sensible data from outside as parameter for `docker run` command. E.g.:

```
> docker run -e PROD_DB=localhost -e PROD_PASSWD=youcannothackit -e
PROD_USER=root myappname
```

More information: https://gist.github.com/Faheetah/a2a401a01d2d56fa7d1a9d7ab0d2831b

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/



Docker Compose

Docker Compose - way to run system in click

If you need to run an application, which consist of several microservices, Compose is for you.

Compose is just an automation tool for running several docker containers by `docker run`

Compose uses .yml (or .yaml) configuration files to setup all needed environment

Moreover, Compose can control Swarm of Docker containers

Composition configuration

Configuration approach

Compose uses YAML files for describing which services will run and how they will set up:

- Network ports mapping (host <--> guest)
- Volumes configuration
- Container-connection networks configuration
- Dependency between container on load
- Many other...

Much more here: https://docs.docker.com/compose/compose-file/

Language of .yml for Compose

```
version: “{version_of_compose_usually_3}”
services:
  {service_name}:
    image: {name_of_contained_from_hub}
    build: {path_to_dockerfile_directory}
    ports:
      - “{host_port}:{guest_port_to_map_for}”
    volumes:
      - {guest_path}:{host_path_to_source}
```

Much more here: https://docs.docker.com/compose/compose-file/

Compose file example

```
version: “3”
services:
  business_logic_server:
    image: bl-service:v1.3
    build: .
    ports:
      - “8080:80”
    volumes:
      - /opt/test_app_resources:/opt/app_resources
```

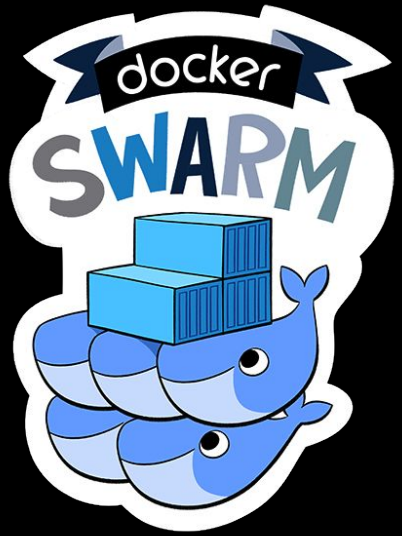
Run, check and stop Compose

```
> cd {path_to_folder_with_yaml_file}

> docker-compose -f {yml_filename} up -d

> docker-compose -f {yml_filename} ps

> docker-compose -f {yml_filename} stop
```

Source and more info: https://docs.docker.com/compose/reference/overview/

Docker Swarm

(Finally!)



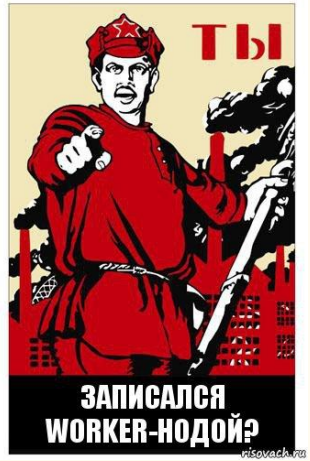
Docker Swarm

Toolkit for creation and orchestration of cluster
for Docker-based systems.

It provides possibility to **automate** distribution of containers between several physical hosts, configuring rules of distribution, network connection between nodes, attached volumes.

Moreover, it allows you to perform **hot** reconfiguration of cluster system.

Join Swarm



All nodes inside Swarm can be “manager” or “worker”.

Manager node will watch for Swarm configuration, do containers distribution, monitoring and load balancing. Create it by command:

```
> docker swarm init
```

Worker node would join Swarm and take instructions from Manager one. Create it by command (with parameters provided after Manager node creation):

```
> docker swarm join --token {token} {manager_ip}:{port_2377}
```

You can also join Swarm as manager to improve fault-tolerance of system.

Source and more info: https://rhelblog.redhat.com/2015/07/29/architecting-containers-part-1-user-space-vs-kernel-space/

Swarm Configuration

Usage of Swarm

For configuring Swarm you need to edit .yml Compose file.

(Compose file version should be 3 or higher)

Compose instructions introduces additional keywords for the compose standard, all of them inside **deploy** class.

Source and more info: https://docs.docker.com/compose/compose-file/#deploy

Adding rules for deployment

```
{service_name}:
  deploy:
    replicas: {number_of_service_replicas_inside_cluster}
    resources:
      limits:
        cpus: {limit_for_cpu_usage_per_service}
        memory: {limit_of_memory_usage_per_service}
    restart_policy:
      condition: {service_restart_conditions}
```

Source and more info: https://docs.docker.com/compose/compose-file/#deploy

Adding placement rules

You can also add specific rules for node selection for current service replicas.

```
deploy:
    # other deploy options
    placement:
      constraints: {constraints_for_placement}
      preferences: {preferences for placement}
```

Source and more info: https://docs.docker.com/compose/compose-file/#placement

Important notes

Image for every service should be available from Docker Hub, hence, you should push your webserver image (after creating personal account) to Docker Hub. Please, keep it public, because it will be needed for grading.

Version of Compose file should be 3 or higher (correspondingly, version of Docker should be 1.13.0 or higher)

build option in Compose file will be ignored while deploying in Swarm mode

Source and more info: https://docs.docker.com/compose/compose-file/#placement

Compose with Swarm example

```
#service configuration
   deploy:
      replicas: 4
      resources:
        limits:
          cpus: '0.30'
          memory: 100M
      restart_policy:
        condition: on-failure
      placement:
        constraints: [node.role == manager]
```

Swarm - start cluster

On master node run following command to start stack of services in Swarm mode (you can **run it on the fly** to update after changes, try it):

```
> docker stack deploy -c {compose_file} {cluster_name}
```

Show list of running services:

```
> docker service ls
```

Stop the cluster with name {cluster_name}:

```
> docker stack rm {cluster_name}
```

Assignment

Single advice:
Think before clicking
Ctrl+V

Assignment

1. Write simple server using Python and Flask (or any other simple web framework you comfortable with), which will return “Hello world” on your request. See next pages for code.
2. Put it into container (build new one using Dockerfile), remember about resolving dependencies (use RUN instruction and pip tool for python).
3. Add hit counter to your code (increasing on every page load), keeping counter inside Redis storage, which will also run in another container (will think about Redis in next steps). See next pages for code.
4. Put code into next version of Docker image (hint: use -t parameter for build command).
5. Run and check that webserver, packed inside container, works.

Assignment

6. Create .yml file for Compose configuration, which will run both your web server and Redis containers, within single network space.
7. Run them and check that everything is working as supposed.
8. Add Swarm capabilities to your Compose configuration and run your containers on two nodes. Increase amount of web-server service replicas. Keep Redis service on Manager machine.
9. Push your web-server image into cloud (needed to Swarm)
10. Run manager node on firstVM and worker node on second VM.
11. Start Swarm on two machines, check that server replies with different hostnames.
12. Add visualizer node for Swarm distribution (image - `dockersamples/visualizer:stable`)

Assignment

13. Update Swarm on the go, check that visualizer shows you service distribution between nodes
14. Reboot worker node and see what will happen.
15. Write report about your findings, adding screenshots with results of your work. Provide descriptions of challenges, which you faced, and findings about creation of containerized cluster.

Final result

1. **Dockerfile** for creating web-server, which can count hits on page using Redis as storage. Web server should start on simple ‘docker run’ command with parameters without further commands.
2. **Compose configuration file 1**, which start one webserver container and one Redis container, making them available for each other.
3. Image of webserver uploaded to public repository at Docker Hub
4. **Compose configuration file 2** for running 6 copies of webserver service, one Redis and one visualizer container on Swarm cluster.
5. **Report** with description of results of your work with Docker, Compose and Swarm and your description of challenges and lessons learned during lab.

Assignment - Web server on Python

```
from flask import Flask
import os
import socket

app = Flask(__name__)

@app.route("/")
def hello():
    html = "<h3>Hello World!</h3><b>Hostname:</b> {hostname}"
    return html.format(hostname=socket.gethostname())

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Assignment - Web server with Redis

```
# other imports
from redis import Redis, RedisError

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)
app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>Redis server disabled, counter is unreachable</i>"
    html = "<h3>Hello World!</h3><b>Hostname:</b> {hostname}</br><b>Visits</b> {visits}"
    return html.format(hostname=socket.gethostname(), visits=visits)

if __name__ == "__main__":
  app.run(host='0.0.0.0', port=80)
```