Equinor Developer Conference 2019
Tuesday 17. September

https://github.com/equinor/edc2019-docker

Docker introduction workshop

K3U

or:

How do we get our stuff out in the cloud already!?


ONE DOES NOT SIMPLY
"GO TO THE CLOUD"
made on imgur

Part 1
Background

- Background of Docker.
- Which problems does Docker solve?
  - Portability
  - Isolation
  - Overhead
  - Mutability
- The limitations of Docker.
- Terms and definitions

Background of Docker

2008: Google created “cgroups”

2013: Docker released

PS: A container is not a virtual machine!

Central Docker concepts

The **Dockerfile** is an imperative definition of how to create a **Docker image**.

The **Docker image** can be run as a **container** on any machine with Docker installed.

Docker improves **portability**

Build once. Run everywhere.

Develop & collaborate without friction.

Simplify operations.


IT WORKS ON MY MACHINE
THEN WE'LL SHIP YOUR MACHINE
AND THAT IS HOW DOCKER WAS BORN
imgflip.com

Docker improves **isolation**

Filesystem isolation for dependencies.

Network isolation simplifies port management.

CPU and memory isolation allows for limiting resource usage.

All of this also decreases the risk of accidental or intentional interference with the host operating system.

Docker reduces **overhead**

	Virtual machine	**Docker container**
Artifact size	200MB to 10GB+	5MB to 500MB
Startup time	1-5 minutes	1-10 seconds
Artifact build time	Minutes to hours	Seconds to minutes

Consequences:

- Shorter development and feedback loops
- Better suitable for horizontal scaling and scale-out architecture
- Easier dynamic behaviour in cloud environments (scaling, updating)

Docker encourages **immutability**

Traditionally: manual upgrades, changes, troubleshooting and fixes.

Now:

Immutable infrastructure is designed to be **replaced** instead of changed.

Immutability allows for (and requires) more **automation** and generally results in **greater agility and stability** if done right.

*Docker and the modern cloud ecosystems embraces the **immutability mindset**, and so should you :-)*

When not to use Docker

Almost any system can be built to run in Docker in a good way.

However, there are probably times when Docker is going to cause **more pain than pleasure**.

Traditional complex monolith with lots of state and tentacles: **Use caution!**


WORKED FINE IN DEV
OPS PROBLEM NOW
memegenerator.net

Part 2
Technical

- Set up Docker.
- Start and interact with a container.
- Read and write a Dockerfile.
- Build and push an image.
- Deploy an image to Azure

Installing Docker

Locally:

https://github.com/equinor/edc2019-docker#preferred---docker-locally

VS Code remote:

https://github.com/equinor/edc2019-docker#alternative---remote-with-vs-code

SSH:

https://github.com/equinor/edc2019-docker#alternative---remote-with-putty

Running and interacting with containers

Run an image

Run an image in the foreground:

```
docker run mysql:8
```

This will fail since MySQL refuses to start without a password configured.

Run a container with a configuration given as an environmental variable:

```
docker run -e MYSQL_ROOT_PASSWORD=yolo mysql:8
```

Run an image - detach & name

To start the container in the background, add `--detach/-d`:

```
docker run -d -e MYSQL_ROOT_PASSWORD=yolo mysql:8
```

Giving the container a name simplifies interaction later:

```
docker run --name mysql -d -e MYSQL_ROOT_PASSWORD=yolo mysql:8
```

Interact with container - logs

View the logs of a container running in the background:

```
docker logs -f mysql
```

Interact with container - commands & shell

Execute a command inside a running container:

```
docker exec mysql ls
```

Start and connect to a shell inside the container:

```
docker exec -it mysql sh
```

Stop and remove a container

Stop a container:

```
docker stop mysql
```

Remove a container:

```
docker rm mysql
```

Run an image - mounting a folder

Run a container with a folder (must exist) on the host machine mounted inside the container (remember drive sharing permissions on Windows):

```
docker run -v c:/edc/mysql:/var/lib/mysql \
--name mysql -d -e MYSQL_ROOT_PASSWORD=yolo mysql:8
```

Run an image - expose network ports

Run an image and expose a network port:

```
docker run -p 8080:80 nginxdemos/hello
```

Try accessing it on http://localhost:8080 !

Building images

Dockerfile

Slightly simplified version of the official MySQL Dockerfile used to build the official MySQL Docker images:

```
FROM oraclelinux:7-slim

ARG MYSQL_SERVER_PACKAGE=mysql-community-server-minimal-8.0.13
ARG MYSQL_SHELL_PACKAGE=mysql-shell-8.0.13

RUN yum install -y https://repo.mysql.com/mysql-community-minimal-release-el7.rpm \
     https://repo.mysql.com/mysql-community-release-el7.rpm \
    && yum-config-manager --enable mysql80-server-minimal \
    && yum install -y $MYSQL_SERVER_PACKAGE $MYSQL_SHELL_PACKAGE libpwquality \
    && yum clean all \
    && mkdir /docker-entrypoint-initdb.d

EXPOSE 3306 33060
CMD ["mysqld"]
```

Building an image

```
docker build --tag mysql:8-stian .
```

The tag consist of a name (mysql) and tag (8-stian).

The `.` at the end is where to search for a file named `Dockerfile`.

After the build is finished the image is available locally and can be seen by doing `docker images`.

Log on to image registry

Images are stored in a ***registry***.

The default registry is at hub.docker.com

To save some time I have created a dedicated registry for the workshop.

`docker` **`login harbor.edc.stian.tech`**

Username: **edc**
Password: **edcEDC3DC**

To log in to hub.docker.com just do `` `docker login` ``. You can be logged in to multiple registries at the same time.

Publish (push) an image

Add a tag including repository information to the image:

```
docker tag mysql:8-stian
harbor.edc.stian.tech/edc/mysql:8-stian
```

Push our image with:

```
docker push harbor.edc.stian.tech/edc/mysql:8-stian
```

Extra credit - Deploy to Azure WebApps

If you have access to an Azure Subscription

Deploy an image to Azure

These steps requires that you have Azure CLI installed.

```
az login
az group create --name docker-playground \
                --location northeurope
```

After creating a “folder” for our resources above we create an App Service “Plan”. A plan is basically just a managed virtual machine with a given size that we can run several containers on top of:

```
az appservice plan create --resource-group docker-playground \
                          --name docker-playground --is-linux --sku B3
```

And finally we deploy an image to the plan:

```
az webapp create --resource-group docker-playground --plan docker-playground \
                 --name nginxdemo --deployment-container-image-name nginxdemos/hello
```

Take note of `defaultHostName` in the output. Try it in your browser :)

Deploy an image to Azure (cont.)

For images that needs configuration env variables can be set like this:

```
az webapp config appsettings set --resource-group docker-playground \
          --name nginxdemo --settings SOME_PASSWORD="sdfljwef"
```

Container needs to be restarted to effectuate the changes:

```
az webapp restart --name radixgrafana --resource-group radix-monitoring
```

To view information about a container use `webapp show`:

```
az webapp show --name nginxdemo --resource-group docker-playground
```

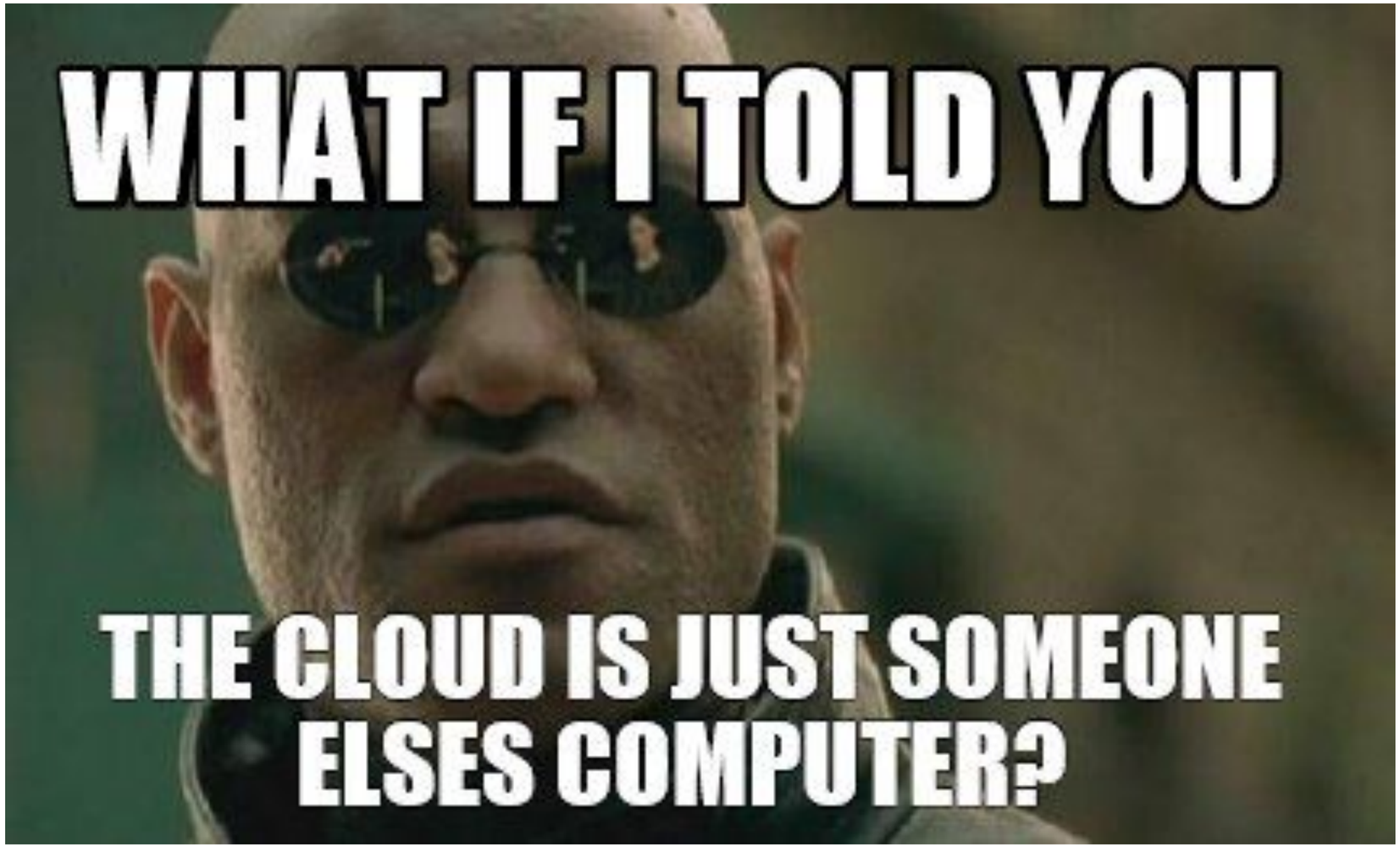
Play

https://github.com/equinor/edc2019-docker

time

Suggestions:

- Play with existing images
- Build simple images, hello world style
- Dockerize your own apps


WHAT IF I TOLD YOU
THE CLOUD IS JUST SOMEONE ELSES COMPUTER?

Some terms and definitions

Docker, container(ize), image are often used interchangeably in discussions and online. They do have different meanings though the context is usually enough to avoid confusion for experienced users.

Docker is a **company**. Docker is also a software to **build** container images. Docker is also software to **run** container images.

Some terms and definitions

A (container) image is the artifact resulting from a Docker build. Docker produces images conforming to the Open Container Initiative (OCI) standard, even though almost nobody talks about OCI itself.

A container is a **running instance** of a container, or Docker image.

There are also other tools that can build images. And other runtimes to run images.

Part 3
Next steps

- Developing for cloud
- Layers, optimizations.
- Multistage builds.
- Gotchas - secrets
- Ways to deploy and operate containers.

Developing for cloud

Even though almost any application can be Dockerized there are several patterns that will make it much easier to deploy and operate.

A set of these patterns is the 12 Factor App which is described on 12factor.net .

Developing for cloud - configuration

Use environment variables for configuration. (https://12factor.net/config)

Building a configuration file in an image makes it un-reusable.

On most deployment options setting environment variables is 10x easier than trying to mount in a file.

Save yourself the pain of using configuration files if possible.

Developing for cloud - logging

Log to standard out. Never to files. (https://12factor.net/logs)

Let the infrastructure take care of collecting and processing logs.

Accessing log files inside a container requires special knowledge of the specific application and can be a real hassle, especially if trying to debug during an outage.

Developing for cloud - persistence

Don’t keep persistent state in the container. (https://12factor.net/processes)

- Need a database? Use Azure MySQL.
- Need a memory cache? Use Azure Cache for Redis.
- Need a queue? Use Azure Service Bus (AMPQ).

A lot of the benefits of containers assumes that they are immutable and quickly replaceable.

PS: When using managed services, try to choose services with an industry standard interface (MySQL, Redis, AMPQ) so that migrating to another provider is easy. Try to avoid using services requiring special integration towards one specific cloud provider.

Dockerfile optimizations - base images

Pay attention to the base image for your Docker images.

The smallest base image that is widely used is `alpine`. **It's under 5MB!**

`debian:stretch-slim` is 55MB and `debian:stretch` is 101MB.

Alpine uses musl instead of gcc so some build processes might need adjustments to build for alpine.

Dockerfile optimizations - layers and caching

For every command in your Dockerfile a filesystem snapshot (layer) will be created.

When building, Docker can see if some commands are unchanged and re-use those layers.

So install dependencies and things that rarely change first and compile/build/test your actual code last.

Several commands can be chained together with `&&` to be part of the same layer. This can also save space if done right.

Dockerfile optimizations - multistage

Multistage builds can be used to create a fat build container using lots of tools and dependencies. The resulting artifact can be copied to a new slim image **without any of the bloat** from the build container. Pseudocode:

```
FROM nodejs AS build
RUN npm install && npm build

FROM nginx:alpine
COPY --from=build /app/dist.min.js /usr/share/www/
```

The build container might be 2-300MB but the resulting web-server + JS file can be **under 10MB**.

Dockerfile gotchas

A Docker image contains **full history** of all commands that have been run and the **filesystem state** after each command.

For example if you copy in a SSH private key using COPY needed to download some dependencies from a private repository the key will **still be stored** in an intermediate layer even if you `rm` it later.

Solution to this is to either download such dependencies outside the Docker build or use **multistage builds**.

Deployment and operations

Now that you have a bunch of **immutable container images** ready how do you easily **deploy and manage** the container instances?

You can use a PaaS such as **Azure App Services**.

You can use **Kubernetes**, which is rapidly becoming the **de-facto standard for container orchestration**. AWS, GCP, Azure, Digital Ocean and more each have managed Kubernetes services that can be set up in minutes.

You can use **Omnia Radix**, a CI/CD platform we are building in Equinor that runs on top of Azure Kubernetes.

Next steps

LearnDocker.online

Equinor Slack channels:

- #docker
- #kubernetes

K30

Thank you for your attention!

Stian Øvrevåge
Solutions architect at K30


twitter.com/StianOvrevage


github.com/StianOvrevage


#docker & #kubernetes


stian.tech