Microservices

What are microservices

- grains of application code that run minimal functionality that can be isolated from other grains of application code
- These different grains are loosely coupled to each other, and the different grains are independently developed as well as maintained
- The central idea behind microservices is that
    - some types of applications become easier to build and maintain when they are broken into smaller composable pieces that work together.
    - Each component is continuously developed and separately maintained, and the application is then simply the sum of its constituent components.

Benefits

- When broken down in pieces, applications are easier to build and maintain. Because they are smaller entities.
- Each piece is independently developed, and the application itself is just a sum of all the pieces.
- Smaller pieces of application code are easier to understand.
- If one component fails, you spin up another while the rest of the application continues to function.
- Smaller components are easier to scale.
- Individual components are easy to fit into continuous delivery pipelines, and complex deployment scenarios.

Why Microservices and Containers are perfect match

What is container

- a container is an application that runs based on a container image.
- The container image is a lightweight standalone executable package of software that includes everything that is needed to run an application.
- containers are lightweight and include all dependencies required to run an application, containers have become the standard for developing applications nowadays.

Why Microservices and Containers are perfect match

- As containers are focusing on their specific functionality, they are perfect building block for creating microservices
- The main requirement is that the different components are connected to each other in the right way
- These static values can be provided in a very flexible way by using container orchestration platforms such as Kubernetes.

Breaking up Monolithic Applications

2 ways of breaking up applications

- Tiers: based on architectural layers
- Services: based on application functionality

Tier based

- determined by how close functions are to end users and how far they are from the data that is stored.
- easy to replace tiers in a flexible way, or to add solutions to a tier.

- Ex. breaking up in presentation, business logic, and persistence layers.

presentation layer	business layer	the persistence layer
exposes the application to the outside world using a web server	the application layer, giving access, for instance, to the company ordering system.	the database server, ensuring that everything is stored properly.

Service based

- complex applications typically consist of different independent services.
- Ex. an online store that is offering a product catalog, a shopping cart, a payment system, order processing, shipping, and more.
- while breaking up monolithic applications, several changes are often applied.
    - Connection parameters to databases and middleware need to be changed from hard-coded to variables that can be managed in a flexible way.
    - In web applications, application calls need to be changed to static public DNS host names.
    - security needs to be modified, to ensure that one application is allowed to access other applications. This is also known as cross-origin resource sharing.
- Breaking up monolithic applications can be made easier using different solutions that are provided in Kubernetes and other orchestration platforms.

How applications communicate

- Synchronous communication: a client sends a request and waits for a response from the server.
    - the threat is blocked until an answer is received.
- Asynchronous communication: the client does not wait for an answer.
    - the client in the meantime can continue and do something else.

Synchronous communication

- HTTP is used
- web based API's are used to enable synchronous communication

asynchronous communication

- a message broker like AMQP or RabbitMQ is frequently used
- the application just sends the request to the message broker and can continue.
- asynchronous API's are used as well, like WSDL.

The role of the API

- The API is all about communication between the different components in microservices.
- An API defines how programs request access to services from either operating systems or all applications.
- RESTful API is common
- gRPC is more efficient

What is RESTful API

- Rest stands for representational state transfer
- it's an architectural style for distributed systems.
- Rest is based on six guiding principles.

    - Client server: the client interface is separated from the server part to increase portability.
    - Stateless: the server keeps no information about the state of the request, and each request must contain all the required information.
    - Cacheable: the server must indicate if data in the response can be cached.
    - uniform interface: The interface to the server is well defined and uniform.
    - layered system: an architecture should be designed as multiple hierarchical layers where a component cannot see beyond its own layer.
    - code on demand: Rest allows client functionality to be extended by downloading and executing code as applets or scripts.
- The RESTful API defines resource methods to perform a desired transition to interact with web services.
- the HTTP GET or PUT or POST or DELETE methods are used for this purpose.
- A key feature is the RESTful API should be entered without prior knowledge beyond the initial URI,
- RESTful API is a big standard in microservices, as it provides an easy to use, stateless and uniform way for different parts of the micro service to interact.

What is gRPC

- gRPC is an API that is based on the Unix remote procedure call model.
- addressable units are procedures and data is hidden behind the procedures, which is opposite to the approach in rest.
- gRPC is using a binary format.
    - makes it easier to provide structure.
- gRPC provides multiplexing which means that multiple sessions can exist in one connection.

REST vs gRPC

REST	gRPC
work with JSON, which is accessible and easy	protbuf messages, which is a very efficient and packed format
format it uses is not that efficient.	mainly based on HTTP version 2.0. And HTTP 2.0 is binary and for that reason, much more efficient.
mainly based on HTTP version 1.1.	HTTP 2.0 uses multiplexed streams, which means that multiple streams can be sent at the same time without making a new connection
HTTP 1.1 is textual, and typically has a lot of overhead	

The Role of CI/CD in Microservices

what is CI/CD

- a method to deliver applications in a flexible and automated way
- it is used because it makes the integration of new code in a solution real easy.

- introduces ongoing automation and monitoring throughout the life cycle of applications, from integration to delivery and deployment.
- used in microservices.

what exactly is their role in microservices?

- it is important to implement changes in an easy and non-disruptive way, because your application needs to continue providing services.
    - CI/CD is used to guarantee that this can happen.
- Process
    - the code typically originates in the Git repository, from which it can be deployed by using automation solutions like Dockerfile or OpenShift, which are hosted in a Git repository as well.
- By hosting everything in Git repositories, it's easy to manage development and version differences.

DevOps and Microservices

- developers and operators work together on implementing new software and updates to software.
- In DevOps, CI/CD pipelines are commonly implemented
    - simple GitHub repositories up
    - advanced CI/CD-oriented software solutions, such as Jenkins and OpenShift.

The purpose of DevOps

- reduce time between committing a change to a system and the change being placed in production. DevOps is microservice-oriented by nature, as multiple, smaller products are easier to manage than one monolithic project.
- to have operators and developers working on dedicated tasks.

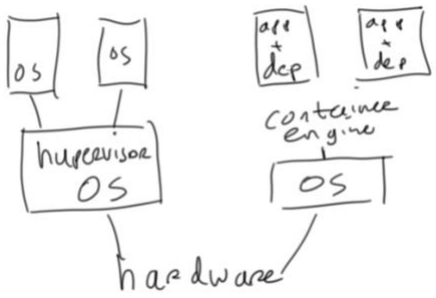
Understanding Containers

What is a container?

- a container is an application with all of its dependencies included.
- Containers always start an application.
    - They don't sit down and wait for you to tell them what to do like a virtual machine.
- To run a container, a container run time is needed.
    - This is the layer between the host operating system and the container itself.

VM vs Container

- Container does not give you OS
    - VM container



Container Types

- System containers
    - behave like virtual machines.
    - go through a complete boot process and provide virtual machines.
    - Typical style services like SSH and logging.
- Application containers
    - used to start just one application.
    - Have become the default.
        - this is what is important in the microservices environment.

Orchestration solutions

- To work with these different container types, like Kubernetes
- focus on managing Application
- typically, don't have a default application to start

Why Containers are Linux

- Containers heavily rely on features that are provided by the Linux kernel
    - Namespaces: an isolated environment.
    - Chroot. the Linux chroot command,
        - is an example of a namespace,
- The Linux kernel can be used as a container runtime in a pure Linux based container environment.

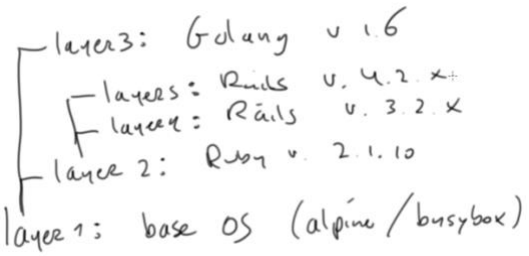
Why using containers make sense in a microservice approach

- Microservices and containers have the same objective in mind
    - microservices focus on developing minimal pieces of code and joining them
    - containers are focusing on developing minimal running application components
- The only thing that needs to be added, is a layer that connects containers together
    - Done by the container orchestration layer

Understanding images and containers

- a container is a running instance of an image.
- the image contains the application called the language runtime as well as the libraries.
- External libraries such as libc are typically provided by the host operating system,
    - but in a container that's different because it is included in the images.
- The container images are read-only instance of the application that just needs to be started.
- While starting a container, the container adds a writable layer on top to store any changes that are made while working with the container.
- A container image consists of multiple layers that are connected together
- Dockerfile is just a script that you are going to run and which will create your own containers.
- Docker images are made up of a series of filesystem layers
- Docker images or container images are made up of a series of file system layers.
    - each layer adds, removes or modifies files from the preceding layer in the file system (overlay filesystem
- By using these different image layers and pointing to other image layers in a smart way, it's easy to build container images that have support for multiple versions of vital components.

- Apart from the different layers, container images typically have a container configuration file that provides instructions on how to run the container.



Tips

1. Try to avoid building large images
2. files that are removed from subsequent layers are still accessible in the base layer, they are just inaccessible.
    a. start from a very essential base layer and build it from there.
    b. if a layer is changed, it changes every layer that comes after it.
        i. changing a proceeding layer means that all subsequent layers need to be rebuilt, repushed, and repulled.
3. order layers from least likely to change, to more likely to change.
4. it's a common mistake to leave all built tools in the application image
    a. avoid this by using multistage built which is a feature that can be used in Dockerfile.
        i. stage one in multistage: generate a Dockerfile with all the compiler stuff in it.
        ii. stage two: copy the generated binary to a working image.

Container Registries

- take care of distribution of images
- two types
    - public (remove registries)
    - private (local registries)

Exploring the container landscape

Container history

- Containers are all about running isolated processes in a protected environment.
- It already started in the 1970's when the chroots system calls was introduced.
- In 2000 freeBSD jails were introduced to partition file systems, network addresses and memory.
- In 2004, Solaris introduced containers to combine system resource control and zone-based boundary separation
- In 2007, control groups were introduced in the linux kernel.
- In 2008, LXC combined cgroups and linux namespaces to implement linux-based containers.
- In 2013, Docker started and became successful by offering a complete container ecosystem.
- In 2014, Kubernetes started as the default container management platform

Container standardization

OCI

- OCI is Open Containers Initiative - a project of the LinuxFoundation.
- Docker has donated its container format and runtime, the runc, to OCI to serve as a solid foundation.
- Established in 2015, with the purpose of creating open industry standards around container formats and runtimes.
    - The Runtime Spec outlines how to run an unpacked filesystem bundle as a container.
        - is basically the specification of the foundation of container images.
    - The image specification outlines how container images should be created.
- because of OCI, different components of the container landscape are highly inter-operable.

CNCF

- CNCF is Cloud Native Computing Foundation. It's a part of LinuxFoundation that promotes the adoption of cloud native computing.
    - Cloud-native computing is an approach in software development that utilizes cloud computing to build and run scalable applications in modern dynamic environments,
        - such as public private and hybrid clouds.
    - technologies such as containers, microservices, serverless functions and immutable infrastructure deployed via declarative code are parts of the cloud native style.
    - Normally, cloud-native applications are built as a set of microservices that run in containers and maybe orchestrated in Kubernetes and deployed using DevOps and Git.
- CNCF hosts the Kubernetes Open source project.

Container Runtimes

What is a container runtime?

- The container runtime is the most important part of containers.
- It is a software that execute containers and manages container images.
- is the most common container runtime, but other container runtimes exists as well.
    - Such as containerd, Rkt and LXD.
- The container runtime can be considered the high level solution to run containers.

Low level container runtimes

- OCI provides standardization for solutions that work with containers like orchestration platforms.
- OCI also has defined container runtime specs.
- Runc is a default implementation of OCI runtime specs,
- The OCI runtime provides low-level functions for running containers.
- The high-level runtime, like Docker and so, provide functions that run on top of the low-level runtime.

Container Runtime Implementations

- The following CRI implementations exist.

    - Dockershim, CRI-O, Containerd, Frakti and rktlet.
- All of these low-level CRI implementations run on top of runc.
- Functionality in the higher level Docker container runtime is being replaced by lower level container runtimes like CRI-O and containerd.

Container Orchestration

- Containers are about just running it. But by just running an application, you don't make it easy to access it, or you don't connect it to other applications.
- To build microservices using containers, additional datacenter features are needed.
    - flexibility and scalability in the networking.
    - Flexible and scalable storage.
    - Methods to connect containers together.
    - Additional services for cluster-wide monitoring of service availability.
- To provide for all of these, an orchestration platform is needed.
- Kubernetes is the standard orchestration platform.

Managing Containers

Setting up docker on Ubuntu or CentOS

Installing docker on Ubuntu

```
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent
software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
add -
sudo apt-key fingerprint 0EBFCD88
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
sudo docker run hello-world
```

Running a Docker Container

Before getting started

- users have to be a member of the docker system group in order to communicate with the Docker daemon allowing them to start and manage containers.
    - Use **sudo usermod –aG docker $(USERID)**
- Do not run container at Root

Running docker containers

- To run Docker containers, Docker registry access must be available
- By default, images are fetched from docker hub
- After finding the image you need you can use Docker run, to start it.
    - Notice, that when starting a system image, parameters must be provided to have it run a default command as well. Like docker run-it busybox /bin/sh.
        - means that a system image is one of the Docker container images that normally that's not common to default application.
    - When you are starting an application image, specifying the name of an application is not required.
        - Docker run -d nginx where -d option
            - to daemonize it.

    - Use exit or Ctrl-p, Ctrl-q to quite/disconnect
    - Use docker ps for an overview of currently running containers

Staging contaienrs

- Use docker run to automatically run a container
- Use docker create to stage a container
- After staging it, use docker start to start it

Stopping Containers

- A container stops when its primary application stops
- Use docker stop to send SIGTERM to a container
- Use docker kill to send SIGKILL to a container
- After stopping a container, it does not disappear
- Use docker rm to permanently remove it

Verifying container availability

Understanding docker run

- If so required, **docker run** will download a container image, and start the main container application as a foreground process
- If no main container application is set, the container comes up and as there is no work to do, will stop immediately
- If a main container application is set, you will get access to this
    - Use **Ctrl-Z, bg** to continue running the container in the background
- To start a system container with a command to run, specify the name of this command while running it, using **docker run -it**
- To run a container with a default application as a background process, use the **docker run -d** command to run in detached mode

Verifying container availability

- **docker ps** gives an overview of containers currently running
- **docker ps --all** also gives an overview of containers that have been running successfully

- Use **docker inspect** to get details about running containers
- Use **docker logs** to get access to the primary application STDOUT
- Use **docker stats** for a Linux **top**-like interface about real-time container statistics

Investigating containers on the Host OS

Docker Components

- Docker CLI is the **docker** command, used by users and resulting in communication with the **dockerd** API
- **dockerd** is the Docker daemon, which listens for Docker API requests and manages the container lifecycle using **containerd**
    - By default, accepts communication through /var/run/docker.sock
    - Alternatively, systemd socket activation can be used for communication
- **containerd** is responsible for managing containers. It manages many aspects, and also runs containers by calling **runc**
- **runc** is the part of **containerd** responsible for running containers and is the OCI compliant part of Docker
- **containerd-shim** allows for daemonless containers, it takes over after **runc** has started the container

Investigating containers on the Host OS

- On the host OS, a Docker container is just a running process that is managed by one of the Docker components
- Because of Linux kernel namespaces, one container does not have access to running parts of the other containers
- When using **ps aux**, all the running Docker containers show as Linux processes

Performing Daily containers

Running containers

Searching for images

- Before running a container, you need to search the right image
  - **docker search ubuntu** will show lots of images
  - **docker search --filter "is-official=true" ubuntu** will only show official images
  - **docker search --filter=stars=30 --filter=is-automated=true centos** will also include appreciation

- Based on what you've found, you can either pull the image, or directly run the container
  - **docker pull centos:centos6**
  - **docker pull --all-tags centos** (will download a LOT)
  - **docker run --rm centos:latest** will pull and run the latest CentOS version (and stop immediately), after which the container is removed
  - **docker images** will show all images locally available

Running contains

- Remember, containers are just a fancy way to run applications
  - **docker run centos** will run the centos:latest image, start its default application and immediately exit
  - **docker run -it centos bash** will run the centos:latest image, start bash, and open an interactive terminal
- Managing foreground and background state
  - **docker run -it centos bash** will run the container in the foreground
    - Use **Ctrl-p**, **Ctrl-q** to disconnect
    - Use **exit** to quit the main application
  - **docker run -dit centos bash** will run the container in the background
  - **docker attach container-name** will attach to the running container if it is started with **-d**
- Since Docker version 1.13, docker container run instead of docker run is the preferred command

Advanced docker run options

- **docker run -d --rm --name mywebserver nginx** will run nginx, and remove the container image after running it

Publishing Ports

- By default, container applications are accessible from inside the container only
- To make it accessible, you'll need to publish a port
- **docker container run --name web_server -d -p 8080:80 nginx** runs the nginx image, and configures port 8080 on the docker host to port forward to port 80 in the container

Managing containers Resource Limitations

Understanding Memory Limitations

- As containers are just Linux processes, by default they'll have full access to the host system resources
- The Linux kernel provides Cgroups to put a limit to this
- **docker run -d -p 8081:80 --memory="128m" nginx** sets a hard memory limit
- **docker run -d -p 8082:80 --memory-reservation="256m" nginx** sets a soft limit, which will only be enforced if a memory shortage exists

Understanding CPU Limitations

- Docker inherits the Linux kernel Cgroups notion of CPU Shares
- If not specified, all containers get a CPU shares weight of 1024
- When starting a container, a relative weight expressed in CPU shares can be specified
  - **docker run -it --rm -c 512 mycontainer --cpus 4** will run the container on 4 CPUs, but with relative CPU shares set to 50% of available CPU resources
- Containers can also be pinned to a specific core using **--cpuset-cpus**
  - **docker run -it --rm --cpuset-cpus=0,2 mycontainer --cpus 2** will run the container on cores 0 and 2 only

Connecting containers

- Different options exist to connect containers
  - Linked containers are a legacy Docker feature that allow containers to share environment variables
  - Shared volumes can be used to share data between containers
  - Docker networks are the current preferred way that allow containers to share information
  - Notice that linking through the Docker network behaves differently when used on a bridged network or another network type

Managing Container Images

Container Images

- Images are what a container is started from
- Base container images are available at hub.docker.com
- Users can upload images to Docker Hub
- Go to hub.docker.com to search for images
- Or use **docker search** to search for images

Layered File Systems

- Container images are using multiple layers of file system
  - If you build your own solution on top of a generic container image, your solution will provide an additional layer that just points to the generic container images
- Docker images are immutable, each modification adds an extra layer to the pre-existing layers
- The container sees it as a single virtual file system by using UnionFS or another driver

Image layers

- **docker image ls** shows images stored
- **docker history <imageID>** or **docker history image:tag** shows the different layers in the image
- Notice that each modification adds an image layer!

Creating Images

- Roughly, there are two approaches to creating an image
- Using a running container: a container is started, and modifications are applied to the container. From the container, **docker** commands are used to write modifications

- Using a Dockerfile: a Dockerfile contains instructions for building an image. Each instruction adds a new layer to the image. This offers more control over whatfiels are added to which layer

Building Images with Dockerfile

Dockerfile

- Dockerfile is a way to automate container builds
- It contains all instructions required to build a container image
- So instead of distributing images, you could just distribute the Dockerfile
- Use **docker build .** to build the container image based on the Dockerfile in the current directory

- Images will be stored on your local system, but you can direct the image to be storedin a repository

Using Dockerfile Instruction

- FROM: identifies the base image to use. This must be the first instruction in Dockerfile
- MAINTAINER: the author of the image
- RUN: executes a command while building the container, it is executed before the container is run and changes what is in the resulting container
- CMD: specifies a command to run when the container starts
- EXPOSE: exposes container ports on the container host
- ENV: sets environment variables that are passed to the CMD
- ADD: copies files from the host to the container. By default files are copied from the Dockerfile directory
- ENTRYPOINT: specifies a command to run when the container starts
- VOLUME: specifies the name of a volume that should be mounted from the host into the container
- USER: identifies the user that should run tasks for building this container, use for services to run as a specific user
- WORKDIR: set the current working directory for commands that are running from the container

ENTRYPOINT and CMD

- Both ENTRYPOINT and CMD specify a command to run when the container starts
- CMD specifies the command that should be run by default after starting the container. You may override that, using **docker run mycontainer othercommand**
- ENTRYPOINT can be overridden as well, but it's more work: you need **docker run --entrypoint mycommand mycontainer** to override the default command that is started

- Syntax

  - Commands in ENTRYPOINT and COMMAND can be specified in different ways
  - The most common way is the Exec form, which is shaped as **<instruction> ["executable", "arg1", "arg2"]**
  - The alternative is to use Shell form, which is shaped as **<instruction> <command>**
  - While shell form seems easier to use, it runs <command> as an argument to /bin/sh, which may lead to confusion

- Ex.

- Dockerfile demo is in https://github.com/sandervanvugt/containers/sandertest
- Use **docker build -t nmap .** to run it from the current directory
- Tip: use **docker build --no-cache -t nmap .** to ensure the complete procedure is performed again
- Next, use **docker run nmap** to run it
- For troubleshooting: **docker run -it nmap /bin/bash**
  - Will only work if you've installed bash!

Building images with docker commit

Managing images with docker commit

- After making changes to a container, you can save it to an image
- Use **docker commit** to do so
  - **docker commit -m "custom web server" -a "Sander van Vugt" myapache myapache**
  - Use **docker images** to verify
- Next, use **docker save -o myapache.tar myapache** and transport it to anywhere you'd like
- From another system, use **docker load -i myapache.tar** to import it as an image

Pushing Images to Registries

Tags

- Tags are used to describe information about a specific image version
- Tags are aliases to the image ID and will show when using **docker images**
- Tags are typically set when building the image:
  - **docker build -t username/imagename:tagname**
  - For private use, the **username** part is optional, when pushing it to a public registry it is mandatory
- Alternatively, tags can be set using **docker tag**
  - **docker tag source-image{:tag] targetimage[:tag]**
- If no tag is applied, the tag **:latest** is automatically set
  - **:latest** always points to the latest version of an image
- Target image repositories can also be specified in the Docker tag
- Tags allow you to assign multiple names to images
  - A common tag is "latest", which allows you to run the latest
  - Consider using meaningful tags, including version number as well as intended use (like **:prod** and **:test**)
- In Centos, all images are in one repository, and tags are used to describe which specific image you want: centos:7.6
- Manually tag images: **docker tag myapache myapache:1.0**
  - Next, using **docker images | grep myapache** will show the same image listed twice, as 1.0 and as latest
- Manually tag images: **docker tag myapache myapache:1.0**
  - Next, using **docker images | grep myapache** will show the same image listed twice, as 1.0 and as latest
- Tags can also be used to identify the target registry
  - **docker tag myapache localhost:5000/myapache:1.0**

Using Tags

Creating Private Registries

- On CentOS
  - **yum install docker-distribution**
  - **systemctl enable --now docker-distribution**
  - Config is in /etc/docker-distribution/registry/config.yml
  - Registry service listens on port 5000, open it in the firewall
    - **sudo firewall-cmd --add-port 5000/tcp**
- On Ubuntu
  - **docker run -d -p 5000:5000 --restart=always --name registry registry:latest**

Ex: using your own registry

- **docker pull fedora**
- **docker images**
- **docker tag fedora:latest localhost:5000/myfedora** (the tag is required to push it to your own image registry)
- **docker push localhost:5000/myfedora**
- **docker rmi fedora**; also remove the image based on the tag you've just created
- **docker pull localhost:5000/myfedora** downloads it again from your own local registry

Automating Image builds from Git Repositories

Autobuild

- Autobuild triggers a new image build when source code has changed in the Git repository
- To configure it, webhooks must be setup from the iamge registry

Configure Autobuild

1. Configure sources in the Github repository
   a) https://github.com/sandervanvugt/containers/sandertest
2. Create the repository in https://hub.docker.com
3. Select Repositories > Reponame > Build > Link to Github and make connection to Github by entering username
4. After making connection, click Github icon and enter build details
   a. The GitHub repository to connect to
   b. (Optional) Autotest
   c. (Optional) Build rules (a default rule is provided)
5. Click Build to run the first build

Best Practice

Keeping Image Small

- Images are multi-layered
- Removing data from later layers doesn't make sense: the data will still be in the base layer
- Use specific base images, and avoid generic images like the ubuntu image
- Consider using multistage builds where a Dockerfile goes through different stages, and only the last stage is kept
- Minimize the number of **RUN** commands in Dockerfile, as each run command creates its own layer
  - Use && to add multiple commands to one RUN command: **RUN apt-get -y update && apt-get install python**
- If you have many images with much in common, consider creating your own base images
- In Kubernetes, use Secrets and Configs to keep data out of the images

Managing Container Storage

Understanding Container Storage

- Container storage by nature is ephemeral, which means that it lasts for a very short time and nothing is done to guarantee its persistency
- When files are written, they are written to a writable filesystem layer that is added to the container image
- Notice that as a result, storage is tightly connected to the host machine

- To work with storage in containers in a more persistent and flexible way, permanent storage solutions must be used

- One solution is to use a bind mount to a file system on the host OS: the storage is managed by the host OS in this case
- Another solution is to connect to external (SAN or cloud-based) persistent storage solutions
- To connect to external storage, Volumes are used, and specific drivers can specify which volume type to connect to

- For temporary data, tmpfs can be used

Understanding Storage Drivers

- Storage drivers allow for writing data in the writable layer of the container
- Data written this way is not persistent and performance is not good
- The writable layer is created when running any container
- Storage drivers handle how the layers interact with one another
- For local use, the **local** driver and the **sshfs** driver are available

- For more enterprise level use, drivers are available through the Kubernetes or Swarm orchestration layer
-

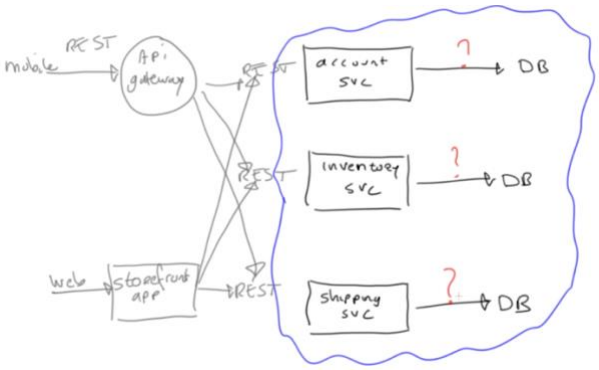
The role of Container Orchestration

Enterprise Container Requirement

- Containers are based on the Container Runtime, a solution that has nothing to connect multiple containers together in a Microservice architecture
- Containers need to be distributed, scheduled and load balanced
- Apart from that, High Availability is required as well as an easy solution to provide updates without downtime
- To run containers as microservices, additional platform components are required as well, such as software defined networking and software defined storage

Microservices Platform Requirements

- To understand microservices platform requirements, you need to understand the typical Microservices application
- Microservices applications typically drill down to a database part and an accessibility part, using REST or Web to provide accessibility



From Microservice to Platform

- Based on the microservice overview on the previous slide, the following minimal platform requirements can be defined
  - Different databases must be running as connected applications
  - Front-end services need to be added to that, and add an accessibility layer as well
  - At the user side, accessibility must be added to different types of user requests
- Add some scalability to this, as well as some high availability, and you'll have the basic platform requirements
- And optionally, add integration of a CI/CD pipeline as well to make the cycle from source code to application complete

Exploring the Container Orchestration Landscape

The Orchestration Landscape

- Kubernetes is the leading solution and the open source upstream for many other platforms
- Docker Swarm was developed by Docker, Inc. and offered as Docker Enterprise - now a part of Mirantis
- Red Hat OpenShift is based on the OpenShift Kubernetes Distribution (OKD) and offers Kubernetes with strong developed CI/CD

The leading position of Kubernetes

- Kubernetes is the leading technology that is in nearly all container orchestration platforms
- This is because of its origins, coming from Google Borg
- Google donated the Kubernetes specifications to the open source community after running Borg internally for over a decade
- As a result, a free, stable and open source project was introduced, where all vendors could base their own solution
- This rocked the world of container orchestration, and resulted in Kubernetes being the leading platform for orchestration

Understanding Kubernetes

Understanding Kubernetes Delivery Options

- Kubernetes as a managed service in public and private cloud
  - Amazon EKS
  - Google Cloud Platform GKE
  - Azure AKS
  - OpenStack Magnum
- Kubernetes as an on-premise installation using a Kubernetes distribution
- Kubernetes as a test-drive platform, mainly developed to learn Kubernetes

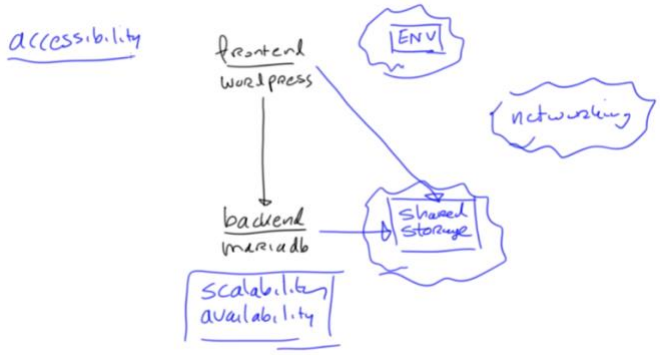
Understanding K8S Distributions

- When using on-premise Kubernetes, something needs to be installed
- Open Source Kubernetes can easily be installed using **kubeadm** on any supported platform
- Kubernetes distributions offer the additional value of providing support and on occasion more stable code
- Most on-premise Kubernetes distributions can be used in cloud as well

Common Kubernetes Distributions

- Canonical Kubernetes: offered by the makers of Ubuntu, relatively pure and based on open source, runs on premise and in cloud
- Rancher: focus on offering Kubernetes as multi-cluster deployments. Available as on-premise and in cloud
- SUSE CaaS platform: offered by SUSE as an on-premise and private cloud solution - support for public cloud is pending
- OpenShift: developed by Red Hat, available on-premise and in cloud. Adds CI/CD features to core Kubernetes functionality

Design K8S

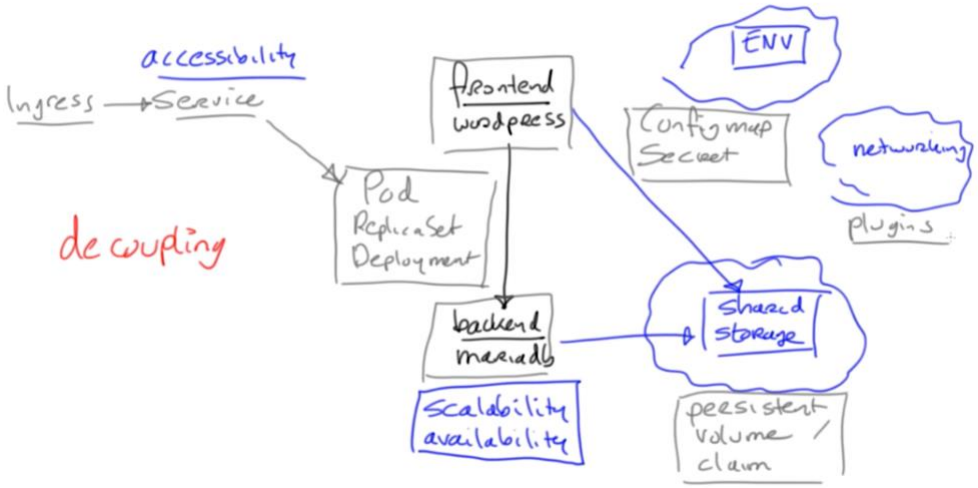


Understanding Kubernetes

Using Kubernetes in Minikube

- Minikube is an all-in-one Kubernetes virtual machine
- Using Minikube is recommended as it gives access to all Kubernetes features, without any limitations

Core Application Components in Kubernetes



Running Containers through Pods and Deployments

- Run containers in a Kubernetes environment within a pod.
- The pod is adding additional properties to the container that make it possible to run the container decently in a scaled multi-node environment.
- Do not run so-called naked pods because pods should be managed and they should be managed by a deployment.
    - don't run naked pods unless it is for testing
- **kubectl run** will run a pod
- **Kubectl create deployment will create a deployment**

Understanding the Philosophy of Decoupling in a Microservices context

Understanding Decoupling in Microservices

- In microservices, it's all about independent development cycles, allowing developers to focus on their chunk of code
- To make reusing code easy, separation of static code from dynamic values is important
- This approach of decoupling items should be key in your microservices strategy
- Kubernetes helps, by adding many resource types that focus on decoupling

Decoupling Resource Types in K8s

- ConfigMap: used for storing variables and config files
- Secrets: used for storing variables in a protected way
- PersistentVolumes: used to refer to external storage
- PersistentVolumeClaims: used to point to the storage type you need to use

Using the kubectl Utility in Declarative or Imperative Mode

- To run containers, the *imperative* approach can be used
    - **kubectl create deployment**
    - **kubectl run**
- Alternatively, the *declarative* approach can be used, where specifications are done in a YAML file
- The declarative approach is preferred in a DevOps environment, as versions of YAML manifests can easily be maintained in Git repositories

Understanding YAMLL

- YAML is a markup language, where parent-child relations between objects are indicated using indentation
- The objects are key: value type objects
- If a key can have multiple values, each value is indicated with a hyphen
- Use spaces for indentation, using tabs is forbidden

- `source <(kubectl copmletion bash)`

Understanding the Kubernetes API

Understanding the API

- The API server provides access to Kubernetes objects
- The API is also the specification of objects and their properties
- **kubectl** is the key utility to write objects to the API
- The Kubernetes API is extensible, which makes it easy to add new objects
- This is continuously happening, which is also why a new version Kubernetes is released every 3 months

Exploring the API

- **kubectl api-resources** will show the different resources, including the API group they come from
- **kubectl api-versions** will show different objects, the API they come from, as well as the current version
- **kubectl explain** allows you to browse through the different objects in the API and explore how they are constructed
    - This is a key skill for discovering how to set up the YAML code where objects are defined

Troubleshooting Kubernetes Applications

Essential Troubleshooting

- **kubectl get** gives an overview of different object types and should be used as first step in troubleshooting
- **kubectl describe** explores Kubernetes resources as created in the Kubernetes etcd database
- **kubectl logs** shows the STDOUT for applications running in a container and is useful for application troubleshooting
- **kubectl events** gives an overview of recent events
- **kubectl get <resource> -o yaml** reveals all status information and is useful for debugging

Creating Container-based Microservice

Feeding Images into Kubernetes

- If no registry hostname is specified, Kubernetes fetches images from the Docker public registry
- To manage images stored in other registries, refer the complete path to the image: **my.registry.example.com/myimage**
- For using images from private registries, authentication may be required

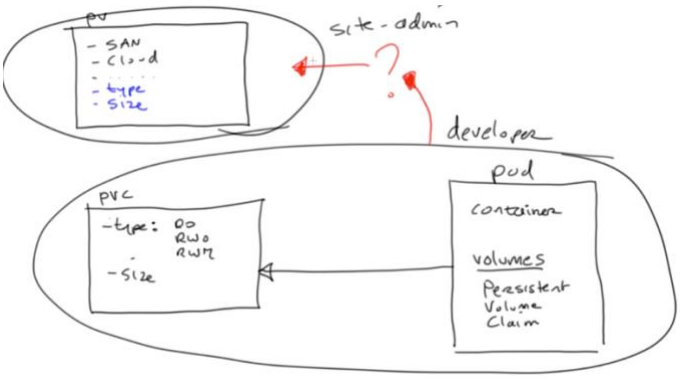
Understanding imagePullPolicy

- By default, images are fetched from registries
- To use local images, use imagePullPolicy: ifNotPresent or imagePullPolicy: Never
    - For working with local images, you'll need to make sure all cluster nodes have local pre-pulled images
    - On the Kubernetes nodes, images are managed by the container layer and typically stored in an image database
    - If the container layer is Docker, use **docker images ls** for a list of images
    - To set up a private registry, use the procedure offered by the container layer

Organizing pod storage

Understanding Pod Storage

- Volumes are a part of the Pod container spec
- For increased flexibililty and decoupling, it is recommended to work with external storage using PV and PVC
- Persistent Volume (PV) is a non-namespaced Kubernetes object that connect to different types of external storage
- Persistent Volume Claim (PVC) contains a specification of required storage needs
- The Pod Volume Spec specifies which PVC to use to connect to a specific type of storage



Using Storage Provisioners

Using Dynamic Storage Provisioning

- PVs can be created manually
- Alternatively, PVs can be dynamically provisioned
- To dynamically provision a PV, a StorageClass must be referred to in the PVC
- Storage classes are using classes that are defined by the administrator
  - It's up to the admin to decide what to use to categorize: storage type, backup type, QoS, or something else
- Storage classes are also using provisioners that connect to a specific storage type
  - Each provisioner has its own parameters
- A default storage class can be used, alternatively storage classes can be defined manually

Understanding Default StorageClass

- A Default storage class can be used to always specify a storage type so that it doesn't have to be defined in the PVC spec anymore
- If no specific storage class is defined, the default storage class is used
- Often, the hostPath storage type is used for this purpose
- Notice that hostPath should not be used in clusters, as continuous access to storage cannot be guaranteed!
- Using the default storage class provided by a cluster is simple: nothing has to be specified in the PVC definition

Managing Container Storage

Understanding Container Storage

- Container storage by nature is ephemeral
  - Lasts for a very short time
  - Nothing is done to guarantee its persistency
- When fields are written, they are written to a writable filesystem layer that is added to the container image
- Storage is tightly connected to the host machine
- To work with storage in containers in a more persistent and flexible way, permanent storage solutions must be used

Understanding storage solution

- Use a bind mount to a file system on the host OS: the storage is managed by the host OS
  - Not flexible
- Connect to external (SAN or Cloud-based) persistent storage solutions

- To connect to external storage, Volumes are used, and specific drivers can specify which volume type to connect to
- For temporary data, tmpfs can be used

Storage Drivers

Understanding storage drivers

- Storage drivers allow for writing data in the writable layer of the container
- Data written this way is not persistent and performance is not good
- The writable layer is created when running any container
- Storage drivers handle how the layers interact with one another
- For local use, the local driver and the sshfs driver are available
- For more enterprise level use, drivers are available through the Kubernetes or Swarm orchestration layer

Selecting a storage driver

- For local use
    - Aufs, overlay, overlay2, btrfs, zfs, devicemapper
- Storagedrivers are set in /etc/docker/daemon.json

CoW Strategy

- Copy on Write (CoW): if a container changes files in a lower layer, the file is copied from the lower layer to the upper layer and modified while it is there
- Using the CoW strategy guarantee that only modifications are stored in the writable filesystem layer, which is very storage efficient
    - If multiple container are started based on one image, only the differences between the container writable layers is stored
- For write-intense applications, this strategy is not so efficient, and it's better to use external storage

Exploring CoW Storage

- Create a directory cow-test: **mkdir cowtest; cd cowtest**
- Create a file hello.sh:
    - #!/bin/sh
    - echo "hello"
- Make it executable: **chmod +x hello.sh**
- Create the file Dockerfile.base:
    - FROM ubuntu:20.04
    - COPY . /app
- Create the file Dockerfile:
    - FROM koe/base-image:1.0
    - CMD /app/hello.sh

- Build the base image: **docker build -t koe/base-image:1.0 -f Dockerfile.base .**
- Build the second image: **docker build -t koe/final-image:1.0 -f Dockerfile .**
- Check image sizes to see the same size is reported for both: **docker image ls**
- Use the image ID as displayed by the **docker history** command to investigate what has happened in each of the layers:
    - **docker history <ID-of-base-image>**
    - **docker history <ID-of-final-image>**
- Notice that the difference is only in the top layer of the second image, for the rest the images are the same and the disk space is used once only

Using Bind Mount as Container Storage

Understanding Bind Mounts

- Bind mount storage is based on Linux bind mounts
    - In a Linux mount, you connect a device like the disk to a directory, so. That the directory is the entry point that allows you to write to the disk
- In a bind mount you connect one directory to another directory
    - As long as the bind mount is on the host operating system, it provides for persistent storage
- The container mounts a directory or file from the host OS into the container
- If the host dir doesn't yet exist, it will be created, but only if the -v option is used
- The host OS still fully controls access to the file
- Docker commands cannot be used to manage the bind mount
- The -v option as well as the –mount option can be used to create the bind mount
    - -v is the old option, which combined multiple arguments in one field
    - –mount is newer and more verbose

Case for Using Bind Mounts

- Bind mounts work when the host computer contains the files that need to be accessible in the containers
    - Configuration files
    - Access to source code
    - Log files

Creating a Bind Mount

- Using **--mount**
    - **mkdir bind1; docker run --rm -dit --name=bind1 --mount type=bind,source="$(pwd)"/bind1,target=/app nginx:latest**
- Using **-v**
    - **docker run --rm -dit --name=bind2 -v "$(pwd)"/bind2:/app nginx:latest**
- Use **docker inspect <containername>** to verify

Volumes

Benefits of Using Volumes

- Preferred way to work with persistent data as the volume survives the container lifetime
- Multiple containers can get simultaneous access to the volumes
- Data can be stored externally
- Volumes can be used to transition data from one host to another
- Volumes are supported for Linux and Windows containers
- Volumes live outside of the container and therefore don't increase container size
- Volumes use drivers to specify how storage is accessed.
    - Enterprise-grade drivers are available in Docker Swarm – not stand alone

Working with Volumes

- **docker volume create myvol** creates a simple volume that uses the local file system as the storage backend
- **docker volume ls** will show the volume
- **docker volume inspect my-vol** shows the properties of the volume
- **docker run -it --name voltest --rm --mount source=myvol,target=/data nginx:latest /bin/sh** will run a container and attach to the running volume
- From the container, use **cp /etc/hosts /data; touch /data/testfile; ctrl-p, ctrl-q**
- **sudo -I; ls /var/lib/docker/volumes/myvol/_data/**
- **docker run -it --name voltest2 --rm --mount source=myvol,target=/data nginx:latest /bin/sh**
- From the second container: **ls /data; touch /data/newfile; ctrl-p, ctrl-q**

Multi-container Volume Access

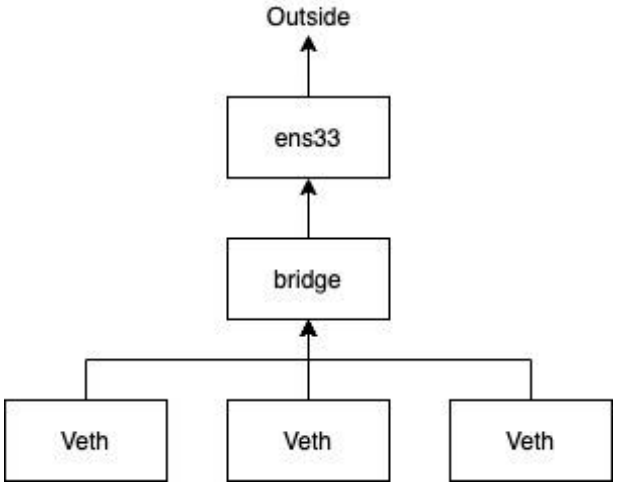
- To just create a file in a volume, nothing special is needed and the volume can be accessed from multiple containers at the same time
- To simultaneously access files on volumes from multiple containers, a special driver is needed
- Recommended: use the readonly mount option to protect from file locking problems
    - **docker run -it --name voltest3 --rm --mount source=myvol,target=/data,readonly nginx:latest /bin/sh**
- Enterprise-grade drivers: Docker Swarm, Kubernetes
- For non-orchestrator use, consider using the local drive NFS type

Container Networking

Understanding Container Networking

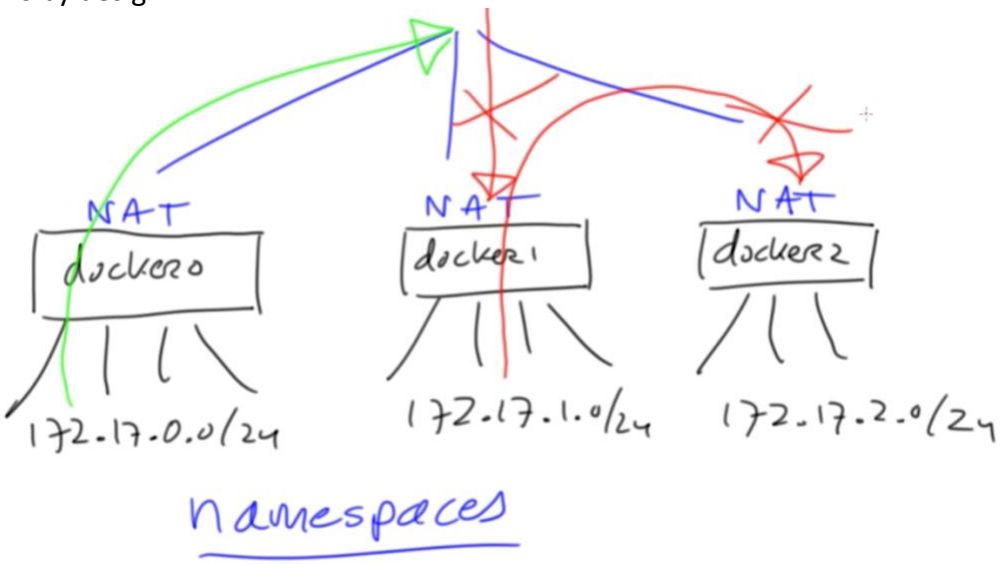
Docker Networking

- Container networking is pluggable and uses drivers
- Default drivers provide core networking
    - Bridge: the default networking, allowing applications in standalone containers to communicate
    - Host: removes network isolation between host and containers and allows containers to use the host network directly.
        - In Docker, only available in swarm
    - Overlay: in Swarm, allows different docker daemons to be connected using a software defined network. Allows standalone containers on different Docker hosts to communicate
        - Not built-in,
    - Macvlan: assigns a MAC address to a container, making it appear as a physical device on the network,
        - Excellent for legacy applications
    - None: completely disable networking
    - Plugins: uses third-party plugins, typically seen in orchestration software
- Diagram
    - Ens33: a physical network card on the host
    - Bridge: a logical device
    - Veth: device created by the container



Understanding Bridge Networking

- Bridge networking is the default: a container network is created on internal IP address 182.17.0.0/16
- Containers will get an IP address in that range when started
- Additional bridge networks can be created
- When creating additional bridge networks, automatic service discovery is added, so that new containers can be reached by name
- There is no traffic between different bridge networks because of namespaces that provide strict isolation
- You cannot create routes from one bridge network to another bridge network and that is by design



Working with Default Bridge Networking

Connecting Containers on the Bridge Network

- Type **docker network ls** to see default networking
- Start two containers on the default network
  - **docker run -dit --name alpine1 alpine ash**
  - **docker run -dit --name alpine2 alpine ash**
- Verify the containers are started
  - **docker container ls**
- Check what is currently happening on the Network Bridge and notice the IP addresses of the containers
  - **docker network inspect bridge**
- View the container perspective on current networking
  - **docker attach alpine1; ip addr show**
- Verify connectivity (from within the container)
  - **ping 172.17.0.3**
- Use **Ctrl-p**, **Ctrl-q** to detach from the alpine container
- Stop and remove the containers
  - **docker container stop alpine1 alpine2**
  - **docker container rm alpine1 alpine2**

Creating a Custom Bridge Network

- Create a custom network
  - **docker network create –driver bridge alpine-net**
  - **docker network ls**
  - **docker network inspect alpine-net**
- Start containers on a specific network. Notice that while starting, a container can be connected to one network only. If it needs to be on two networks, you'll have to do that later
  - **docker run -dit --name alpine1 --network alpine-net alpine ash**
  - **docker run -dit --name alpine2 --network alpine-net alpine ash**
  - **docker run -it --name alpine3 alpine ash**
  - **docker run -dit --name alpine4 --network alpine-net alpine ash**
  - **docker network connect bridge alpine4**
- Verify correct working
  - **docker container ls**
  - **docker network inspect bridge**
  - **docker network inspect alpine-net**
- Verify automatic service discovery, which is enabled on user defined networks
  - **docker container attach alpine1; ping alpine4**
- But notice this doesn't work on the default bridge
  - (still from alpine1) **ping alpine3**
- There's no routing either:
  - (still from alpine1) ping 172.17.0.2
- But all containers can reach out to the external network

Understanding Microservices Container Networking Needs

- In microservices, connected containers may be running on different hosts
- To allow these to directly connect, overlay networking is required
- Overlay networking is not a part of default container stacks, but included in orchestration solutions like Docker Swarm and Kubernetes

- Diagram
    - Tunnel: VPNs