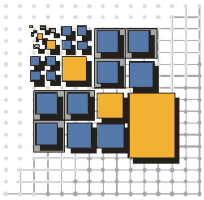


Otto-Friedrich-University Bamberg

Distributed Systems Group

Service Oriented Architecture (SOA)

Assignment 2

Introduction and Description
(Summer term 2020)

Preface

Virtualization is the most important engineering enabler in today's cloud computing and service hosting world. Software architects build, package and ship their applications in reusable units like containers on a containership.

Virtual machines (VMs) started the rise of cloud computing with first offerings like AWS EC2, where customers can rent VMs dependent on their needs. Since VMs have some well known shortcomings like long start-up/boot times, a more lightweight solution was needed to respond within a few seconds/milliseconds. Containers are currently the answer to face these problems.

> "This containers revolution is changing the basic act of software consumption.
> It's redefining this much more lightweight, portable unit,
> or atom, that is much easier to manage...
> It's a gateway to dynamic management and dynamic systems."
> - Craig McLuckie, former Google - now Heptio -

Docker is the most prominent platform to build containers based on your application. These containers make it easy to scale your application by increasing the number of running containers, port your containerized application to other hosting environments, e.g. change the cloud service provider, etc. But how to orchestrate them? A single container is not that helpful when building a microservice architecture with dozens of services.
Kubernetes (K8s - abbreviations comes from the 8 characters between K and s), currently the most prominent container orchestrator, can handle this for us. As a survey[1] shows, Docker (1st place) and Kubernetes (3rd place) are technologies people want to work with!

So let's get started with this practically highly relevant topics, but before we dig into the technologies, a few literature hints for Docker and K8s. Since both ecosystems change frequently, the literature here is as up to date as possible:

- **Docker Documentation** - I believe, you do NOT need a book about docker, when reading the documentation. So give it a try and follow the links to topics we provided in the text. Otherwise, here is the general docker documentation page `https://docs.docker.com/`
- **Nigel Poulton: Docker Deep Dive: Zero to Docker in a single book** - Nigel is a Docker Captain (comparable to Java Champions in the Java world). His book is a good source when you want to read something about docker (buy the e-book, this version is frequently updated).
- **Nigel Poulton: The Kubernetes Book** - Also frequently updated - not that mature as the docker book, but a good source.

[1] The survey was conducted in 2020 at StackOverflow: Docker 24.5% and Kubernetes 18.5%. `https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-platforms-wanted5`

Contents

Prerequisites - Installation Remarks

Installing Docker is somehow a challenge of choosing the right version and setting. For *Docker Desktop* you need a specific Windows OS build, also Enterprise or Pro version. There is also the possibility since 2020 to use an up to date version of Windows Home. You can see and check the prerequisites on the Docker Desktop page[2]. If you fulfill these prerequisites, you can also use Docker Desktop, a single node Kubernetes cluster can be enabled and used.

Because I assume, that not all of us fulfill the prerequisites, we use Docker Toolbox. Docker Toolbox[3] is marked as legacy and also a bit slower but works fine on all Windows versions since it uses VirtualBox for virtualization.

For Kubernetes, we use Minikube[4] as a single node cluster. HINT: If you installed Docker Desktop, you enable Hyper-V which caused problems with Minikube, therefore, if you use Docker Desktop, use the embedded Kubernetes feature of Docker Desktop.

[2] `https://docs.docker.com/docker-for-windows/install/`

[3] Docs: `https://docs.docker.com/toolbox/toolbox_install_windows/`
Releases: `https://github.com/docker/toolbox/releases`

[4] `https://kubernetes.io/docs/setup/learning-environment/minikube/`

1 Docker

1.1 Overall

Empowering App Development for Developers is the title of Docker's homepage[5]. As already told in the lecture, container technologies helps you to abstract your local environment by decoupling development and operation of your apps. Finding good sources of information which cover the most important topics about docker is hard since a lot of people provide great content with only partial views, but we tried to collect a few really worth reading sources.

A good overview over the different parts like the docker engine[6], dockers architecture, which is also depicted in Figure 1, images, containers and many more aspects of the Docker universe can be found in the official docker get started tutorial[7].

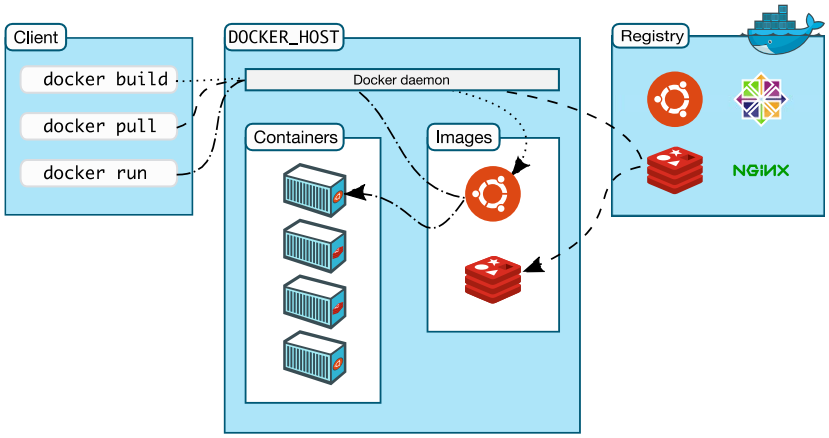


Figure 1: Docker architecture

"A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. [...] When you use the `docker pull` or `docker run` commands, the required images are pulled from your configured registry. When you use the `docker push` command, your image is pushed to your configured registry."[8]

1.2 Best practices

"Small images are faster to pull over the network and faster to load into memory when starting containers or services. There are a few rules of thumb to keep image size small:

[5] `https://www.docker.com/`

[6] As a recap the docker engine consists of a server where the docker deamon is running, a REST API for accessing it, and a CLI client (*docker* commands). The deamon creates all objects like images, containers, volumes etc.

[7] `https://docs.docker.com/get-started/overview/`

[8] Copied from the docker get-started page.
Link to DockerHub: `https://hub.docker.com/`

- Start with an appropriate base image. For instance, if you need a JDK, consider basing your image on the official `openjdk` image, rather than starting with a generic `ubuntu` image and installing `openjdk` as part of the Dockerfile.
- Use **multistage builds**. For instance, you can use the maven image to build your Java application, then reset to the tomcat image and copy the Java artifacts into the correct location to deploy your app, all in the same Dockerfile. This means that your final image doesn't include all of the libraries and dependencies pulled in by the build, but only the artifacts and the environment needed to run them. [...]
- If you have multiple images with a lot in common, consider creating your own base image with the shared components, and basing your unique images on that. **Docker only needs to load the common layers once, and they are cached.** This means that your derivative images use memory on the Docker host more efficiently and load more quickly. [...]
- When building images, always tag them with useful tags which codify version information, intended destination (`prod` or `test`, for instance), stability, or other information that is useful when deploying the application in different environments. Do not rely on the automatically-created `latest` tag.

[...] Avoid storing application data in your container's writable layer using storage drivers. This increases the size of your container and is less efficient from an I/O perspective than using volumes."[9]

Restrict your containers to use only a portion of the available resources to avoid out of memory situations and starvation of processes[10].

1.3 Networking

Docker facilitates some form of isolation via the Linux kernel features, especially `cgroups` and `namespaces`. But how can multiple independent containers on the same host talk to each other? We need some sort of networking, which is secure, but enables this kind of interaction.

There are different network drivers, most importantly the `bridge` driver since it is the default driver and automatically created. This driver enables communication within the host between containers within the same bridge network, but excludes the docker host and containers in other bridge networks (!). "When you start Docker, a default bridge network (also called `bridge`) is created automatically, and newly-started containers connect to it unless otherwise specified. You can also create user-defined custom bridge networks."[11] The `host` driver enables communication within the host between containers, but includes

[9] Some quotes from the docker best practices page. There are many more good hints. Recommendation to read the others: `https://docs.docker.com/develop/dev-best-practices/`

[10] For more details and the options to configure these limits, see the docs: `https://docs.docker.com/config/containers/runmetrics/`

[11] `https://docs.docker.com/network/bridge/`
What's also quite interesting is, that an user-defined bridge (not the default bridge) provides automatic DNS resolution. So you can connect to a container called `frontend` via `frontend` and not specific link rules, which are hard to handle and manage over time.

the docker host. `Overlay` networks connect multiple docker daemons to form some sort of a cluster (first step to enable Docker Swarm mode). `None` disables networking. There are many others available via plugins, but these are the most widely used[12].

Networking Problems Default `bridge` network does not support DNS resolution via container names. This default network is not recommended for usage in production scenarios. "On user-defined networks like alpine-net, containers can not only communicate by IP address, but can also resolve a container name to an IP address. This capability is called automatic service discovery."[13]

By default, when creating a container, it does not publish any port to the outside world (as seen the container:port is only accessible in the corresponding network).

1.4 Storage

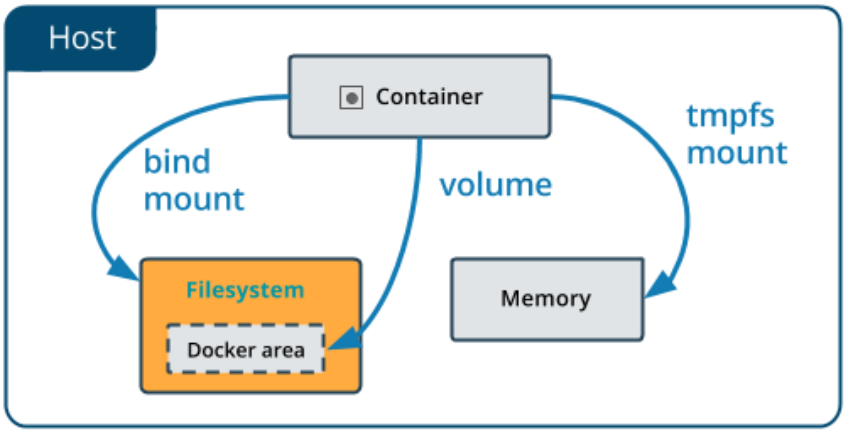


Figure 2: Storage in Docker - Volumes and Bind Mounts

Figure 2[14] shows the (persistent) storage handling in Docker. We only discuss volumes and bind mounts here.

"By default all files created inside a container are stored on a writable container layer. This means that:

- The data is not persisted when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.
- A container's writable layer is tightly coupled to the host machine where the container is running. You can't easily move the data somewhere else.
- Writing into a container's writable layer requires a storage driver to manage the filesystem. The storage driver provides a union filesystem, using the Linux kernel. This extra abstraction reduces performance as compared to using data volumes, which write directly to the host filesystem.

[12] `https://docs.docker.com/network/`
[13] `https://docs.docker.com/network/network-tutorial-standalone/`
[14] `https://docs.docker.com/storage/images/types-of-mounts-bind.png`

Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container is removed: volumes, and bind mounts. [...]

- **Volumes are stored in a part of the host filesystem which is managed by Docker** (/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.
- Bind mounts may be stored anywhere on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time. [...]

1.4.1 Volumes

If you are developing new Docker applications, consider using named volumes instead. You can't use Docker CLI commands to directly manage bind mounts.

If you use either bind mounts or volumes, keep the following in mind:

- If you mount an empty volume into a directory in the container in which files or directories exist, these files or directories are propagated (copied) into the volume. Similarly, if you start a container and specify a volume which does not already exist, an empty volume is created for you. This is a good way to pre-populate data that another container needs.
- If you mount a bind mount or non-empty volume into a directory in the container in which some files or directories exist, these files or directories are obscured by the mount, just as if you saved files into /mnt on a Linux host and then mounted a USB drive into /mnt. The contents of /mnt would be obscured by the contents of the USB drive until the USB drive were unmounted. The obscured files are not removed or altered, but are not accessible while the bind mount or volume is mounted"[15]

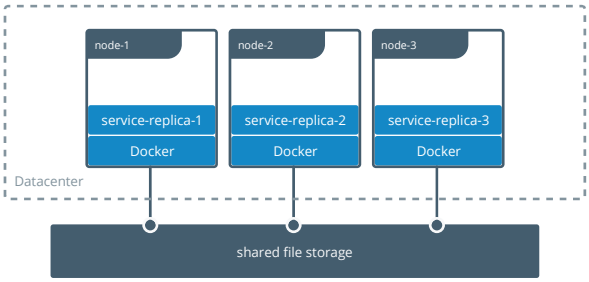
"For some development applications, the container needs to write into the bind mount so that changes are propagated back to the Docker host. At other times, the container only needs read access to the data. Remember that multiple containers can mount the same volume, and it can be mounted read-write for some of them and read-only for others, at the same time."[16]

Share data among machines Use a volume driver for replicated services to access the same files and data in a fault tolerant way. You can for example bind volumes to your replicas which are created via a volume driver and reside for example at Amazon S3.

1.4.2 Bind Mounts

Mounts are more flexible (but cannot be used when creating services), but also more challenging and to some extent dangerous. When binding mounts, you directly have access

[15] `https://docs.docker.com/storage/`
[16] https://docs.docker.com/storage/volumes/



to the file system of your host and this mount is NOT controlled by the docker engine. (So you also have no CLI commands for creating or deleting the folders etc.). Especially read-only access when reading configuration files etc. might be beneficial, but be aware of the consequences when altering these files on the host.

1.5 Multi Stage Builds

Multi stage builds are the docker answer to keep images small. Back then, a lot of operation people spend hours to add and remove the right files to keep images small.

As the name implies, you can have multiple stages (two is NOT the limit). Only the last stage is responsible for creating the image.

```
# base image - builder stage
FROM openjdk:11.0.7-jdk AS builder
# Environemnt Variable
ENV APP_HOME=/root/dev/beverage
# Working directory
WORKDIR $APP_HOME
# Copy all the stuff (easiest way)
COPY . $APP_HOME
# Run the build
RUN ./gradlew build

# base image for the final image (java runtime environment is sufficient)
FROM openjdk:11.0.7-jre
# specificing work directory
WORKDIR /root/
# only copy the fat jar, which includes all dependencies (only a java runtime environment is needed to run it)
COPY --from=builder /root/dev/beverage/build/libs/beverage-all.jar .
# Run it
CMD ["java","-jar","beverage-all.jar"]
```

1.6 Configuration - Environment Variables

Configuration[17] is the ingredient making your application flexible. Think about the `config.properties` file in our REST project, where we specified the `serverUri`. When running your app, you can change this property and your service is exposed on a different port or path without rebuilding your whole application (you don't need to recompile the Java classes). This was really nice and flexible, but now with docker and the multi-stage

[17] The twelve factor app is a document including twelve factors how to build dynamic web apps. It summarizes to some extend best practices. One of these best practices is configuration: `https://www.12factor.net/config`

builds, we face a problem. The configuration file is included in the `jar` file and to change the configuration, we have to rebuild the container image.

But when deploying our containers in different environments which might require different configurations we don't want to rebuild our container images each time. Therefore we need to put the configuration into the environment independent of the container image.

And is there a better place for storing environment configurations than in environment variables? Docker takes this into account and provides a cli option: `env`[18] to provide environment variables to a container. This is commonly used to provide configurations to containers, for example in our case with:

```
docker run -env SERVER_URI=localhost:9090/v1 ...  image
```

The convention for naming environment variables is like constants in Java: uppercase + underscores. In our `config.properties` file, the convention is *camelCase*. Now, we have two different configuration parameters, so which one to choose?

We have to implement an ordering. So the content of our `config.properties` is somehow static (it is included in the jar). The content of our docker environment variable might not be present, but is more flexible, so the ordering is as follows: If a docker environment variable is present, use it, otherwise use the default in some static config files within the image.

In the presented case, you have to implement the configuration handling by hand using `System#getenv` method, but you can also use specialized components for this which enable more clean code using annotations, for example Eclipse MicroProfile Config[19].

[18] You can also use environment variables in your Dockerfile and overwrite it at runtime: `https://docs.docker.com/engine/reference/builder/#env`

[19] `https://www.eclipse.org/community/eclipse_newsletter/2017/september/article3.php`

1.7 Docker Compose

Build multi-container apps by specifying the containers, images, volumes and networks in one declarative file called `docker-compose.yml`. As a hint, install some docker support for your used IDEA, this makes editing Dockerfiles and Compose files a lot easier. Name your containers if possible, and specify the driver for your network.

```yaml
version: "3.0"
services:
  a1:
    image: alpine
    container_name: a1
    stdin_open: true
    tty: true
    networks:
      - a-net
  bev:
    build:
      context: .
      dockerfile: Dockerfile-multi
    container_name: bev
    networks:
      - a-net
    depends_on:
      - a1
    volumes:
      - a-vol:/data
volumes:
  a-vol:
networks:
  a-net:
    driver: bridge
```

Figure 3: A sample compose file

Use version 3 of docker compose as shown in the Figure 3 and check the commands via the docker compose documentation on your own.[20]

[20] `https://docs.docker.com/compose/compose-file`

2 Kubernetes

2.1 Minikube

Since hosted Kubernetes clusters are quite expensive, we use minikube[21] instead. The docs of minikube are also quite nice, check them out[22].

Minikube is a local, single node Kubernetes cluster, which supports the latest Kubernetes versions, it is deployed in a VM, works with docker and provides a bunch of helpful functionalities for users. Per default, a single host path is mounted into the minikube VM, check the docs[23].

Follow the installation guide of the documentation or use Chocolatey under Windows to install minikube and the kubernetes-cli.

Read our GitHub tutorial carefully. This part of the document is shorter since the information are all provided in the tutorial: `https://github.com/johannes-manner/k8s-soa-example`.

2.2 Kubernetes Documentation

The only source of truth in this rapidly changing Kubernetes world, is the official documentation. (And I assume that also this documentation does not cover all the latest changes.)

But: It is the best and most up to date source. So check out the concepts page, where you find also declarative examples of configurations for deployments, replica sets, storage and many more: `https://kubernetes.io/docs/concepts/`.

Tool versions We used Docker 19.03.8, Minikube 1.11.0 and Kubernetes CLI 1.18.5 to implement the tutorial and also for recording the video.

[21] `https://github.com/kubernetes/minikube`
[22] `https://minikube.sigs.k8s.io/docs/start/`
[23] `https://minikube.sigs.k8s.io/docs/handbook/mount/`

3 Assignment Description

DEADLINE: Sunday, 9th of August

Our beverage store implementation was a huge success. Therefore, we need a portable and scalable solution to host our service since a single monolithic application running on one of our laptops is not sufficient any more. This requires some changes in our code and the solution design.

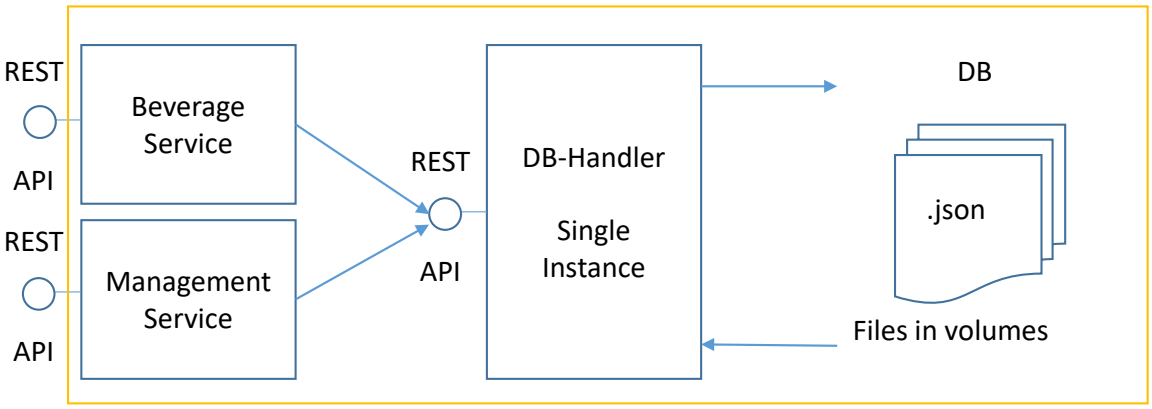


Figure 4: Pictogram of our Assignment 2 architecture

Figure 4 shows a high level view of our new system architecture. We build a *microservices architecture*, where all the boxes are simple JAX-RS compliant Java microservices. They communicate via API calls[24]. We extract our in-memory database and use files instead, which are stored on our host system. The orange border marks the Docker/Kubernetes boundaries, so the microservices and the files should be included and managed by these frameworks. Only the `Beverage Service` and your `Management Service` expose an API, which is accessible outside of the *inner* network.

Specification of your services

- **Beverage Service** - Java Microservice - Re-implement a simple version from the first assignment. The only endpoint this service should be offering is to GET all the beverages, which are stored in DB. Consume the API of the `DB-Handler` to achieve this goal. Your service should be accessible outside of the docker network with port 8080.
- **Managment Service** - Java Microservice - This is the view of your employees, who create, read, update and delete beverages. So re-implement these four facilities from the first assignment. Your service should use the handler and be accessible outside of your docker network with port 8090.
  HINT: Think about different representations (DTOs) of your beverages for your `Beverage Service` and your `Management Service`.

[24] You have to consume your own DB-Handler REST API. Look at `WebTarget` and other classes to do so.

REMARK: The order stuff, which you implemented in the first assignment is not included in this assignment for sake of simplicity[25].

- **DB-Handler** - Java Microservice - A single instance of this DB-Handler should be deployed in both orchestration versions. This is for consistency reasons, since the DB handler reads and writes the files exclusively. Also think about concurrency issues (we will mark them in this assignment). For each request, a single resource instance (runtime automatically instantiates the resource class) is created and serves the request. So race conditions, lost updates etc. can occur. Think about a good design of your solution. The REST API of your DB-Handler should be accessible only within the docker network under `db:9999/v1` (design the API accordingly).
- **DB** - File System - We don't want to store the data in memory, therefore we use files instead. (**REMARK:** in a business scenario, we would use a NoSQL database, but this storage solution let us understand how storage is handled in Docker and Kubernetes). Use appropriate storage concepts, where the corresponding framework handles this for you.

PART 1: Docker All of the described services should be packaged inside their own Docker container and orchestrated via Docker Compose. Make sure to set appropriate dependencies and links to allow an automatic bootstrapping of the service composition via `docker-compose up`. Also support environment variables in your compose file (implement it by hand or via MicroProfile Config) so that service configurations can be easily changed without rebuilding the container images. Especially the connections from the beverage service and the management service to the db handler must be dynamically configurable. If the system is not running after executing `docker-compose up` this part will **not be assessed!**

PART 2: Kubernetes Use the docker containers from PART 1. You should not rebuild your images, make the projects configurable and specify the environment variables via ConfigMaps. For the database part, you are allowed to rebuild the image, if you decide to use StatefulSets (as shown in the example on GitHub) with a mongo db integration, or store the data in files as in the docker case. If the system is not running after executing `kubectl apply -f .` this part will **not be assessed!**

Feel free to explore the container universe These are the minimal requirements you have to implement. They are mandatory! If you have other clever ideas how to make the API even better, contribute and implement it. (There might be some bonus points on the road :)

Running your solution Please provide a meaningful README.md, where you specify the steps, which are necessary to get your solution running. If there is no README, I will use the instructions mentioned above. If the systems are not running, I will not assess them!!

[25] Please inform me, if I should include this again :)

Hints/Remarks

- For sake of simplicity, we simplified our logic compared to the first assignment.
- We know that there is no security mechanism implemented, so malicious users could also alter the orders from other people if they have their ids (but that's not a problem for now).
- Use proper logging where possible. This helps you to find bugs and me to review your assignment.
- Design your DTOs properly (JAXB). Also think about mandatory and optional fields etc.
- Return customized error messages, which include the cause of the error and a descriptive message. Use the correct error class and suitable error status codes.
- Do not use any imperative command. Neither for Docker nor Kubernetes.

Artifacts You have to submit the following artifacts via your group's Git repository.

- Source code of your implemented services (`Beverage Service`, `Management Service` and `DB-Handler`).
- All necessary configuration files (e.g. docker-compse.yml, k8s.ymls ...)
- An Insomnia[26] project, where you "test" your endpoints (make the host configurable).

[26] `https://insomnia.rest/`