DV1566 - Labb 1

Group	17
Student 1	Isac Svensson
Student 2	Robin Stenius

Part 1: Getting started

Q1: Why do you run the container docker/getting started in detached mode?

You do this because you want to not have it attached to any specific console open on the machine, instead having it run in the background. Having it tied to a console could mean accidentally shutting down the console and thus the container as well.

Q2: What is the difference between a container and a container image?

A container image is like a blueprint for a container, containing everything from filesystem information to environment variables and default commands to run once a container is started. While a container is an isolated instance (running process) of a container image, a sort of sandbox environment where the running process leverages kernel namespaces and cgroups.

Part 2: Sample application

Q1: What is the meaning of the docker's directives used in the docker file? Please comment on each of the directives

The meaning of a Docker's directives in a Dockerfile is to tell the Docker how to *build* a certain image, which is the starting point for the custom Docker image later to be used for the container creation.

The **FROM** directive represents the parent image for the current custom Docker image to be built. For example, a certain operating system or database system.

The **LABEL** directive is used to add additional metadata to the Docker image, such as the name of the author, creation date, contact information, etc.

The **RUN** directive is used to execute commands during the image build time. For example, it can be used to install necessary packages, users and user groups which the image might depend on to function properly.

The **CMD** directive is used to command a sort of default initialization which the created container will run once created from the Docker image. Only one CMD command can be executed by the Dockerfile, and if multiple exists, only the last one will be executed.

The **ENTRYPOINT** directive is similar to the CMD directive, in that it is used to provide a default initialization command which will be executed when creating a container from the Docker image. However, the ENTRYPOINT directive *cannot* be overruled if providing command-line parameters. Such as when running the **docker container run** command with the **--entrypoint** flag with provided arguments.

The **ENV** directive is used to set certain environment variables for the image, such as the PATH environment variable, which tells the process which directories to search for executable files.

The **ARG** directive is used to define variables which a user can pass during build time. It is also the only directive that can occur *before* the **FROM** directive.

The **WORKDIR** directive is used to specify the current working directory of the Docker container. This makes it so that any following **ADD**, CMD, **COPY**, **ENTRYPOINT** and **RUN** directives are executed from the given workdir. If it does not exist, it will be created and used as the current working directory.

The **COPY** directive can be used to copy directories and files from the local hosting machine for which needs to be used by the image during the build process. Could be anything from source code files to artifacts.

The **ADD** directive is similar to the **COPY**, except that it can also take a URL as a source parameter for things to be included in the image building.

The **USER** directive can be used to bypass the default usage of the root user for a Docker container, by providing a specific user for the image. The specified user will be used to run all following **RUN**, **CMD**, and **ENTRYPOINT** directives, which may help security concerns.

The **VOLUME** directive is used to specify a Docker volume to be created in the container, mapping this volume inside the container to a directory created in the underlying host machine.

The **EXPOSE** directive is used to tell Docker that the container will be listening on a certain port during runtime.

The **HEALTHCHECK** directive is used to check whether the containers are running *healthily*, for example, checking if the application is actually running within the Docker container.

The **ONBUILD** directive is used to create a reusable Docker image which can itself become the parent for another Docker image.

Q2: Why is it important to tag a container image?

Because tags can be used to display necessary information about an image according to your own needs and personal preference. The tag is basically an alias for the image.

Q3: Why should we bind a host port with the container port?

We should bind a port from the container with the host so that we have a way to reach whatever application(s) that is running in the container. This sets up a firewall exception to forward the port.

Part 3: Update the application

Q1: Is it possible to bind two containers on the same host port?

Yes, it is possible given that there are different networking addresses for each duplicated port which the containers are listening on.

Q2: Why, after stopping a container, you need to remove it?

Because changes made during the container runtime would still be saved after stopping the container since it is not actually deleted. If the container were to be started again, assumptions that it was a *clean* state would be false. A lot of metadata is also stored by Docker, and a lot of "junka" data could accumulate over time and take up unnecessary space.

Q3: Is it possible to remove a running container, without stopping it before the removal?

It is possible, given the "**-f**" flag, which forces it to. '**docker rm -f <id>**'.

Part 4: Share the application

Q1: Given a container image available on a docker image repository, can you start an instance of the image on any docker host? Is there any limitation?

If the host supports Docker, the image will start the same on said host. Anonymous users can only request 100 container images per six hours, while regular (free) users can request 200 container images per six hours.

Part 5: Persist the DB

Q1: If you run two instances of the same container image, let's call them container A and container B, and you create a file in container A, is that new file visible in container B?

Only if they are attached to the same volume, can they share data between each other.

Q2: While in the docker command "docker run -d ubuntu bash -c "shuf -i 1-10000 -n 1 -o /data.txt && tail -f /dev/null" we need to keep the container running with "tail -f /dev/null"?

Because otherwise the container would stop running after outputting the randomized numbers to the given file. This way, the container is still running and the content can be checked.

Q3: What can you do with the "docker exec" command?

You can run commands in a running container, with various options for how the command shall be run, for example in detached mode, read or set environment variables.

Q4: Let assume you need to use a volume. Why do you need to mount the volume in the container filesystem? Does that mean you modify the container filesystem?

The reason to mount a volume to a container filesystem is to store data across multiple instances of containers or between multiple containers. Mounting a volume to a container does indeed modify the container filesystem.

Q5: In this part of the tutorial, you have created a volume. Where is the volume located in the file system of your docker host?

Volumes are stored in a part of the host filesystem which is managed by Docker.
/var/lib/docker/volumes/ on Linux.
\\wsl$\docker-desktop-data\version-pack-data\community\docker\volumes on Windows.

Part 6: Use bind mounts

Q1: What is a dev-mode container?

A dev-mode container is a way to easily collaborate work-in-progress code with your team members. Allow you to switch between your developer environments or your team members' environments, move between branches to look at changes that are in progress.

Q2: Can we run a container, install software dependencies, and then use the updated container without building first the image?

Yes. You can run an image and make changes to the container such as installing dependencies, updating database etc. But, if you then remove the container you will lose all changes and data (unless your data is saved on a bind mount or volume) since you've not updated the image.

Part 7: Multi container app & Part 8: Use Docker Compose

Q1: What is a Docker service?

Running one or more Docker containers with orchestration. Like a master-slave relationship.

Q2: Can we spin up a single instance of a docker container using a docker-compose file?

Yes, this is possible with the command '**docker-compose start <workername>**'.
But depending on the situation different commands may be needed to perform. For example, to start a service with changes this is one way to perform:

1. docker-compose stop -t 1 <worker>
2. docker-compose build <worker>
3. docker-compose up --no-start <worker>
4. docker-compose start <worker>

Q3: Can we run a MySQL container and store the database structure and data in a volume?

Yes, we can store both the data and db-structure in a volume. You can also use a bind mount. If you don’t use either of these options your data won’t be preserved when shutting down container

Q4: Is the order of the services defined in a docker-compose file important, or is it irrelevant which service is defined first?

No, if you need a specific order of starting the services you need to specify that with depends_on, links, volumes_from, and network_mode: "service:..."

Q5: Is it mandatory to define the network in a docker-compose file?

No, by default compose sets up a single network for the application. Where each container for the service joins this default network and is reachable both by the other containers in this network and also discoverable by them at a hostname identical to the container name.

Q6: If you would like to run a multi-container app, is it necessary to use docker compose (i.e. to define a service) or can you achieve the same objective using docker commands from the shell? Does a service offer more than just running a multiple container app with a single command?

No it is not necessary to use docker compose to run a multi-container app.
The service offers more; handling of the containers such as creating, starting, stopping, moving, or deleting. Portability between colleges becomes smoother since you can share a file with the predetermined environment.