Docker

Installation

The following steps are for installing Docker Engine on a Debian system. The instructions are taken from the official documentation.

Install using apt

1. Set up Docker's Apt repository.

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

# Add the repository to Apt sources:
echo \
  "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/debian \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

2. Install the Docker packages.

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

3. Test if docker is working by running

```
sudo docker run hello-world
```

Post-Installation steps

Make it possible for docker to be run without sudo.

1. Create the `docker` group (if necessary).

```
sudo groupadd docker
```

2. Add your user to the `docker` group.

```
sudo usermod -aG docker $USER
```

3. Log out and log back in. If you are running in a VM you might need to restart the VM.
4. Verify that `docker` can be run without `sudo`.

```
docker run hello-world
```

See also

- https://docs.docker.com/engine/install/debian/
- https://docs.docker.com/engine/install/linux-postinstall/

Container setup

Overview

A lot of services will be installed using docker containers for ease of use. This way services can be swapped, upgraded or changed almost at will while still being able to store persistent data easily. Performing backups will be also made easy, since the containers themselves don't need to be backed up.

Docker compose

Docker services are usually run by one of two ways. Either by using the `docker run` command or by using `docker compose`. While both ways have their pros and cons, using `docker compose` is often much more comfortable.

`docker compose` takes a file called `docker-compose.yml` and runs the services that are configured in it. The file can include one or more services (often forming an app-stack), that can each be configured together with additional settings like volumes or networks. It is possible to put every single service you want to use inside a single `docker-compose.yml` file or alternatively create seperate files for seperate app-stacks.

The following shows a small `docker-compose.yml` file for an example app.

```
services:
  app1:
    image: app1
    ports:
      - "8080:80"
    volumes:
      - ./config:/config
      - /path/to/persistentdata/app1:/var/log
    environment:
      - hostname=${HOSTNAME}
    depends_on:
      - redis
  redis:
```

```
    image: redis
```

The compose file specifies two services: `app1` and `redis`. We can see some different things like persistent storage via docker volumes, environment variables with `.env` or port mappings.

With `- "8080:80"` the port 80 inside the container will be mapped to the port 8080 outside the container. If for example the app starts a webserver that internally runs on port 80, then it can be accessed over port 8080 outside the container.

For a more detailed overview see the docker compose documentation.

.env

Some settings can be stored outside of the `docker-compose.yml` file and insted inside the `.env` file. This way, repeating settings like a database username or specific port can be specified at a single point, making later changes much easier.

An example `.env` file can look like this:

```
DB_USER=dbusername
DB_PASS=secretpassword
APP_PORT=9876
```

In the `docker-compose.yml` file, these variables can be accessed by simply writing `${DB_USER}`. When starting the container, the variable gets substituted by the entry in the `.env` file.

More information can be found in the docker documentation.

Persistent Storage

For persistant storage of data (e.g. documents, pictures, videos, etc.) we can use docker volumes. A volume can be used like this:

```
services:
  app1:
    volumes:
      - ./config:/conf
```

Here, the `config` folder, which sits right beside the `docker-compose.yml` folder outside the container, will be mapped to the `conf` folder inside the container. This lets the container read and write (usual permissions apply) to the folder, making it possible to persistently store data. When the container

gets destroyed, all data in the `config` folder will be left untouched. The volumes will be created by docker if they do not exist when starting up the container.

Docker Networks

By default, if no additional configuration is specified, docker automatically creates a single default docker network for all services in the compose file. For better configuration possibilities we can specify a different docker network for the services to use. We can also set some things like the type of network, the subnet used by the network or if IPv6 should be supported.

To specify a different network we can for example write the following:

```
networks:
  default:
    name: bookstack
    driver: bridge
```

This creates a new network called `bookstack` that uses the default `bridge` network driver. By using `default`, every container, unless set otherwise, will be put into the newly created network.

By creating a network by ourselves we can make it possible for other services, that may be created somewhere else, to join the network

See also

- https://docs.docker.com/compose/features-uses/
- https://docs.docker.com/compose/environment-variables/env-file/
- https://docs.docker.com/compose/compose-file/07-volumes/

Tips & Tricks

Simplify docker compose files

When working with a larger app-stack, that uses multiple different services, a lot of times different settings get reused. To simplify we can use yaml anchors and aliases.

Imagine we have the following servies:

```
version: "3.4"
services:
  app1:
    image: app1
    networks:
      - net1
      - net2
    restart: unless-stopped
  app2:
    image: app3
    networks:
      - net1
      - net2
    restart: unless-stopped
  app3:
    image: app3
    networks:
      - net1
      - net2
    restart: unless-stopped
```

We can see, that the restart policy and the networks are repeated for every single service. We could simplify like this:

```
version: "3.4"
x-app_default: &app_default
  networks:
```

```
    - net1
    - net2
  restart: unless-stopped
services:
  app1:
    <<: *app_default
    image: app1
  app2:
    <<: *app_default
    image: app3
  app3:
    <<: *app_default
    image: app3
```

Starting from compose version 3.4 docker ignores top-level keys that start with `x-`. In the example above, wherever we write `<<: *app_default`, everything given after `x-app_default: &app_default` gets inserted.

If a key-value pair is specified in `app_default` and also in one of the services, the `app_default` values get overwritten entirely.

For more information see here.

Docker aliases

The following are examples for aliases, that can be used to start or stop apps.

```
alias dcup='docker compose up -d'
alias dcdn='docker compose down'
alias dcl='docker compose logs -f'
```

To use these, simply add them to your `~/.bashrc` and log out and back in or alternatively use `source ~/.bashrc`.

See also

- https://medium.com/@kinghuang/docker-compose-anchors-aliases-extensions-a1e4105d70bd