Inception

Docker

is a software framework, for building, runnning and managing containers on servers and the cloud.

Same as shipping containers and how they make cargo easily transported by train truck or ship.

Allows the web app to be easily moved between different environments.

How does Docker work?

Docker containers share the host machine's kernel. But they run as isolated processes in their own user space on the host operating system.

They are similar to VMs but with some important differences.

Both VMs and Docker containers provide an isolated environment for running applications, with their own file system, networking and process space.

This allows you to run multiple containers on the same host (docker compose), each with its own configuration, without interfering with each other.

Docker does not include a full copy of the host OS. Instead, they use the host machine's kernel and rely on the host for system calls.

Key differences between VMs and Docker

1. Operating System Isolation

   VMs run on a full copy of an OS, including a separate kernel.

   This provides a high level isolation between the host machine and the virtual machine, as the virtual machine has its own operating system and can run different software than the host machine.

   Docker containers, share the host machine's kernel and rely on the host for system calls.

   This means that they don't have their own OS, but run as isolated processes (click for juicy details) within the host operating system's user space.

   This however provides less isolation than VMs but it also makes docker containers lighter and faster.

2. Resource Utilization:

   Since **VMs** run a full copy of the OS they require a large amount of system resources such as memory and disk space.

   Docker containers share the host machine resources, but they have their own isolated process space and file system. This allows Docker to provide resource isolation and management, making it easier to allocate resources to individual containers.

3. Deployment and Portability:

   Virtual machines are typically deployed as standalone units, with each VM running its own copy of the host OS, which makes it more challenging to deploy and manage multiple virtual machines in a large scale environment.

   Docker containers on the other hand can be easily deployed as a group using Docker compose for example. This makes them easier to manage and deploy apps in a cloud environment or move apps from one host to another.

Understanding Containers

Can be devided to three parts:

- Builder: a series of tools used to build a container, examples would be:

  - Distrobuilder for LXC (Linux Containers): image building tool.
  - Dockerfile for Docker: build docker image.
- Engine: an app used to run a container. (refers to the **docker** command and the `dockerd` daemon)
- Orchestration: technology used to manage many containers.

Containers are a runnable instance of an image. You can create, start, stop, move or delete a container using the Docker API.

You can connect a container to one or more networks, attach storage to it, or create an image based on its current state.

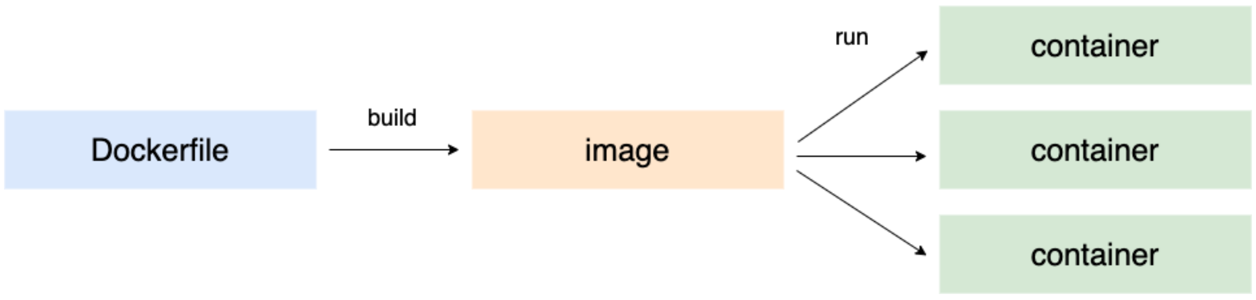
A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Okay, but what are containers?

You can think of Docker containers as a shipping container for software. Just like a shipping container it **holds** all kinds of ‘products’ and can be **loaded** into ships, trains or trucks and **transported** across the world. Similarly a Docker container is a unit that can **hold** software and can be **deployed** across different environments.

How does this relate to what I’m doing here?

Using docker containers is like using Lego blocks to build a complex structure, in our case we’re required to build web app, that has three services: a web server (nginx), a database (mariadb), and the interactive part with the user (wordpress). These services are the building blocks to the app, and need to be contained in seperate different containers. and In order to create those building blocks (just like the diagram bellow) we need what we call ‘**images**’.



Images

To keep the Legos analogy, in order for us to create a lego block, we need a **blueprint** or a **mold** so we could know what or how that block would look like. So images are templates that we use to create containers in whatever parameters or characteristics we need.

Analogies aside, Docker images are the blueprints of docker containers, they are read-only that contains a snapshot of a Docker container. They are a binary file that inclues all the dependencies and configurations required to run an application. In consists of:

- A base operating system
- Application code
- Any dependencies or libraries required to run the application.

Images can also be stored in a Docker registry, such as Docker Hub, where they can be shared and use by others.

How could I create an image?

In order to create an image, we execute or ***build*** a Dockerfile, which specifies the OS, software dependencies and configuration settings. You may be asking what is a Docker..file..?

Now, dockerfiles!

PS: I'll be using nginx as an example here.

A Dockerfile is a text document that contains all the commands we would use to build our image.

Docker reads the instructions given in the file and will use them to automatically build our image.

an example of a Dockerfile would be something like this:

```
FROM debian:buster

RUN apt-get update && apt-get upgrade -y && apt-get install nginx

COPY index.html /usr/share/nginx/html

EXPOSE 443

CMD ["nginx", "-g", "daemon off;"]
```

Above is a dockerfile that would build an image that will run an Nginx web server on port 8080, serving a custom 'index.html' file.

Here's a breakdown of each instruction:

1. `FROM debian:buster`**:** Specifies the base image that this docker Image will be build on top of. In this case, its the offician Debian Buster base image.
2. `RUN apt-get update && apt-get upgrade -y && apt-get install nginx`: This instruction runs commands inside the Docker container to update the package list, upgrade any existing packages, and install the Nginx web server.
3. `COPY index.html /usr/share/nginx/html`: This instruction copies the `index.html` file from the Docker build context (the directory where the Dockerfile resides) to the `/usr/share/nginx/html` directory inside the container. This is where Nginx looks for the default webpage to serve.
4. `EXPOSE 443`: This instruction informs Docker that the container will listen on port 443.
5. `CMD ["nginx", "-g", "daemon off;"]`: This instruction specifies the default command to run when the container is started. In this case, it starts the Nginx web server and keeps it running in the foreground with the `daemon off;` option.

Now we have a Dockerfile, so how do we create an image?

By executing this command, a Docker image is created from the Dockerfile given.

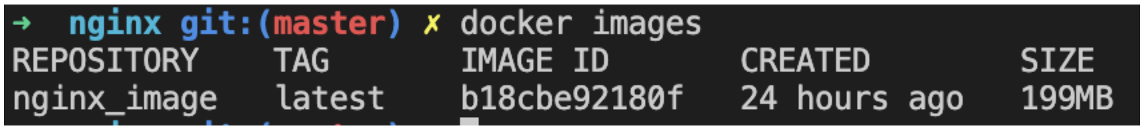
```
docker build . -t <new_image_name> .
```

- -t : is used to give the image a name.
- . : is used to specify the current directory (MUST NOT FORGET)

Then we can verify if the image was created:

```
docker images
```

The output would be something like this:



```
➜  nginx git:(master) ✗ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
nginx_image   latest    b18cbe92180f   24 hours ago   199MB
```

After building an image, now we use it to create a container:

The command needed to do that:

```
docker run -dp 443:443 nginx_image
```

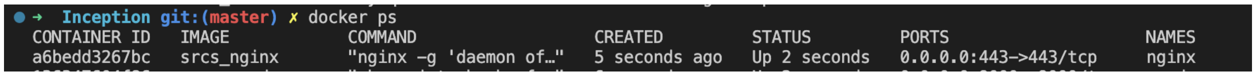
- -d: stands for detached, which means that our Docker container is running in the background of our terminal.
- -p: stands for publish, which is a way of mapping our running container port (443) to the host port (443).

This would output something like this:

```
993bbfcd3c7c913e3bef595ba389237250e381c031a153df9ce5506f3ddf03f4
```

This is what we call a Container ID, and it's a unique identifier that Docker generates for each container instance. It is used to manage the container, such as stopping or starting it, inspecting it's status, or attaching to it's console (which I'll show you how to do it shortly)

We can run this command `docker ps` to verify if the container is running.



```
● ➜  Inception git:(master) ✗ docker ps
  CONTAINER ID   IMAGE         COMMAND                  CREATED         STATUS         PORTS                  NAMES
  a6bedd3267bc   srcs_nginx    "nginx -g 'daemon of…"   5 seconds ago   Up 2 seconds   0.0.0.0:443->443/tcp   nginx
```

Now we can attach to the container's console by executing this command:

```
docker exec -it nginx bash
```

- `-it`: enables an interactive terminal to the container.
- `nginx`: name of the container
- `bash`: tells docker to open a Bash shell inside the container

When the command is executed the container terminal is open. You can use this as a way of testing commands or overall environment before adding the instructions to the Dockerfile.

Now, off to the serious stuff!

Now you know what Docker, Images and containers are, as you may have noticed I used NGINX as an example, which begs the question, what the heck is NGINX?

NGINX

is an open source web server that is now also used as a reverse proxy, HTTP cache, and load balancer.

reverse proxy: sits in front of a web server and receives all the requests before they reach the origin server. Reverse proxies are typically used to enhance performance (find out how by clicking there)
, security, and reliability of the web server.

HTTP cache: a mechanism that allows web browsers and servers to temporarly store or cache certain resources (such as images, scripts, and stylesheets) on a client's device. For performance gains.

Load balancer is a device or software that distributes incoming network traffic across multiple servers.

To put it simply, NGINX can be thought of as a concierge in a hotel. When guests arrive at the hotel, the concierge greets them and directs them to the appropriate rooms. Similarly, Nginx greets incoming requests to a server and directs them to the appropriate web application or service running on the server.

Just as a concierge can handle multiple guests at the same time, Nginx can handle multiple incoming requests and direct them to the correct destination simultaneously. And just as a concierge can be customized to provide different services or amenities, Nginx can be configured to serve different web applications or services on a server, based on the incoming request.

In order to setup an nginx web server an nginx configuration file is needed:

NGINX Server configuration

Similar to Dockerfiles, an nginx configuration file is like a recipe book that the chef uses to make a dish, nginx in this sense is the chef, and you may have guessed nginx.conf is the recipe book. It specifies how **nginx** should behave, what it should do when it receives requests, and how it should respond to those requests as sets of instructions, these instructions include things like which port to listen on, which servers to proxy requests to and how to handle different types of requests.

All Nginx configuration files are located in `/etc/nginx/` directory.

The primary configuration file is `/etc/nginx/nginx.conf`.

NGINX configurations are setup by:

1. directives - they are NGINX configuration options, examples:
    - user,
    - worker_processes,
    - error_log,
    - pid
2. Blocks (also known as contexts) - Groups in which directives are organized.

NOTE: any character after `#` in a line becomes a comment. And NGINX does not interpret it.

An example of an nginx.conf file:

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
       . . .
}

http {
       . . .
}
```

1. `user nginx`: this sets the user that Nginx will run as. It's set to the **nginx** user.
2. `worker_processes 1;`: This sets the number of worker processes that Nginx will use to handle incoming requests. Each worker process can handle multiple connections simultaneously, and having worker processes can improve performance in high traffic scenarios, but for the example's sake its set to 1.
3. `error_log /var/log/nginx/error.log warn;`: This sets the lcoation and level of the error log file. In this case, errors will be logged to **/var/log/nginx/error.log** and only warnings and errors will be logged.
4. `pid /var/run/nginx.pid;`: This sets the location of the nginx process ID file.

   SIDE NOTE: a PID file is a small file containing the process ID of a running program, which is typically used to allow other programs or scripts to interact with the running program. In this case, Nginx creates a file at the given path containing its process ID so that other programs or scripts can interact with Nginx.
5. `events {...}`: This section sets the parameters for Nginx's events model. This includes things like the number of connections Nginx can handle and how long it should keep idle connection pages open.
6. `http {...}`: This section sets the parameters for the HTTP protocol. This includes things like the server name, the location f the server's root directory, and the configuration for individual server blocks.

Location Root and Index Configuration

`root` and `index` determine the content of associated `location` directive block.

Example:

```
location / {
    root html;
    index index.html index.htm;
}
```

in this example, the document root is located in the `html/` directory. Under the defaullt installation prefix for the NGINX, the full path to this location `/etc/nginx/html/`.

Now that we got this off our way..

I ended up using a `default.conf` file instead of the `nginx.conf` one, there is no big difference between them.

The difference in a nutshell would be, the nginx.conf file is the main configuration file for Nginx, while default.conf is a configuration file for a specific Nginx block, which is a way of grouping together configuration settings that apply to a particular set of requests. It contains settings such as server name, liste, and location directives, which apply only to the specific server block that the file is associated with.

Here is my `default.conf` file:

```
server {
    listen 443 ssl;
    root /var/www/html/wordpress;
    index index.php;

    ssl_certificate /etc/ssl/certificate.crt;
    ssl_certificate_key /etc/ssl/private.key;
    autoindex on;
    ssl_protocols TLSv1.2 TLSv1.3;

    location / {
    try_files $uri $uri/ =404;
    }

    location ~ \.php$ {
    include snippets/fastcgi-php.conf;
    fastcgi_pass wordpress:9000;
    }
}
```

1. `listen 443 ssl;` **:** This line indicates that the server should listen on port 443 for HTTPS traffic and use SSL encryption (I'll explain it bellow).
2. `root /var/www/html/wordpress;` : This sets the root directory for the server to `/var/www/wordpress` , where the Wordpress site's files are located.
3. `index index.php` : This line tells the server to use **index.php** as the default index file if a directory is requested.
4. `ssl_certificate /etc/ssl/certificate.crt;` and `ssl_certificate_key /etc/ssl/private.key` : These lines specify the SSL certificate and key file to be used for HTTPS encryption.
5. `autoindex on;` : This enables automatic directory indexing, so that users can browse directories and files on the server.
6. `ssl_protocols TLSv1,2 TLSv1.3` : This specifies which SSL protocols to allow for HTTPS connections.
7. `location / {...}` : This block of code handles requests to the root URL of the server (i.e **https://example.com/**), and uses the `try_files` directive to try to find the requested file or return a 404 error if it cannot be found.
8. `location ~ \.php$ {...}` : This block of code handles requests for PHP files (files ending with .php), and passes them off to the FastCGI server running on the '**wordpress**' container at port 9000. The `include` directive is used to include the `fastcgi-php.conf` file, which contains the FastCGI configuration for PHP. (more on why we need these lines, fastCGI and wordpress at the appropriate sections)

SSL Certification

is a digital certificate that authenticates a website's identity and enales an encrypted connection.

It keeps the connection secure and prevents others from reading o modifying information transferred between two systems.

It works by ensuring that any data transferred between users and websites, or between two systems remains impossible to read.

It uses encryption algorithms to scramble data in transit, which prevents hackers from reading it as it is sent over the connection.

The process (handshake) works like this:

1. A browser or server attemps to connect to a website secured with ssl.
2. The browser or server requests that the web server to indentify itself.
3. The web server sends the browser or a server a copy of its SSL certificate in response.
4. The browser or server checks to see whether it trusts the SSL certificate in response.
5. The web server then returns a digitally signed acknowledgement to start an SSL encrypted session
6. Encrypted data is shared between the browser and the web server.

To generate a TLS 1.2 or TLS 1.3 certificate using OpenSSL, you can use the `genpkey` and `req` commands, similar to how you would generate a self-signed certificate.

1. Generate a private key:
   `openssl genpkey -algorithm RSA -out private.key`
2. Generate a certificate signing request (CSR):
   `openssl req -new -key private.key -out csr.pem`
3. Generate a self-signed certificate:
   `openssl x509 -req -days 365 -in csr.pem -signkey private.key -out certificate.crt`

Once we have the ssl certificate, we need to configure our nginx server.

We need to add the following modifications to the Dockerfile:

```
COPY certificate.crt /etc/ssl/

COPY private.key /etc/ssl/
```

Then, the following lines to the nginx configuration file:

```
listen 443 ssl;

ssl_certificate /etc/ssl/certificate.crt;
ssl_certificate_key /etc/ssl/private.key;

ssl_protocols TLSv1.2 TLSv1.3;
```

NOTE: 443 port is used for HTTPs connections, which use the SSL/TLS protocols.

Now that we setup the web server, we move on to the database.

The database specified in the subject is **MariaDB.**

What is MariaDB?

MariaDB is a fork of mySQL, more or less like an enhanced, drop-in replacement version of MySQL.

A drop-in replacement means that you can substitute the standard mySQL server with the analog version of the MariaDB server and take full advantage of the improvements in the MariaDB without the need to modify your application code.

It supports more storage engines that MySQL, and includes mmany plugins and tools that make it versatile for lots of use cases.

MariaDB Configuration

Create a Dockerfile to setup the MariaDB container.

```
FROM debian:buster

RUN apt-get update && apt-get install -y mariadb-server

COPY ./tools/script ./script

COPY ./tools/50-server.cnf /etc/mysql/mariadb.conf.d/

RUN chmod +x script

RUN ./script

CMD ["mysqld"]
```

The Dockerfile is pretty similar to the nginx one with some key differences which are:

- The installation of `mariadb-server`.
- Copying `script` into the root of the container, you may be wondering what do we need the script for. So when setting up the database we've got to create one to begin with, and a database user, which shouldn't contain the word admin/Admin or administrator/Administrator or any variation of the word and finally we must to modify the root to require a password.

  My script is as such:

```
service mysql start
mariadb -u root -e "CREATE DATABASE <your db name> ; CREATE USER '<your username>'@'%' IDENTIFIED BY '<your password>'
```

```
; REQUIRE NOT REGEXP 'admin|Admin|administrator|Administrator'; GRANT ALL PRIVILEGES ON <db name>.* TO '<username>'@'%'; FLUSH PRIVILEGES;

exec $@
```

1. `service mysql start`: This command starts `mysql`.
2. `mariadb -u root -e "..."`: This command runs mariadb as the user '**root**' and by using the flag **-e** we're able to execute the SQL queries given.

   If you're running the container for testing purposes and wanna try this, just use the `mariadb -u root` command. It'll open mariadb for you and then you can test the queries as you want. (Don't forget ';' at the end of the query ;)).
3. `CREATE DATABASE <db name>;`: This query creates a database with the given name (replace the value between brackets with your db name).
4. `CREATE USER '<username>'@'%' IDENTIFIED BY 'password';`: This query creates a database user with the given name and the given password.
5. `REQUIRE NOT REGEXP 'admin|Admin|administrator|Administrator';`: This is what I used to prevent the creation of a user with the given words as required in the subject.
6. `GRANT ALL PRIVILEGES ON <db name> TO <username>'@'%'; FLUSH PRIVILEGES;` : is used to grant all available privileges on a database to a user. This means that the user will have full access to manipulate, modify and delete data.
7. `ALTER USER 'root'@'localhost' IDENTIFIED BY '<new root password>'; FLUSH PRIVILEGES;`**:** This command is used to modify the root password.
8. `exec $@`: This is usually used to replace the current shell process with the command specified in the argument, but in our case if you look at the docker file `CMD ["mysqld"]` needs to execute after `RUN ./script` is executed, that's why we added that line at the script, otherwise the CMD in Dockerfile woulnd't run.

Now that the database is up and running we move on to Wordpress.

What is Wordpress?

Wordpress is a content management system (CMS) that allows you to host and build websites. It contains plugin architecture and a template system so you can customize any website to fit your needs.

Back with the kitchen analogies, Wordpress can be thought of as a chef's kitchen. Just as the chef's kitchen has all the tools and equipment necessary to prepare and cool meals, Wordpress has all the tools and plugins necessary to create a functional website.

The Wordpress Dockerfile would look a little like this:

```
FROM debian:buster

RUN apt-get update && apt-get install -y php7.3 \
php7.3-fpm php7.3-mysql mariadb-client \
nginx curl

# Downloading WP-CLI
RUN curl -O https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar

# Making the wp-cli.phar file executable and moving it to the /usr/local/bin/ path
# and renaming it 'wp' to use that as the command and not 'wp-cli.phar'
RUN chmod +x wp-cli.phar && mv wp-cli.phar /usr/local/bin/wp

COPY ./tools/script.sh /script.sh
COPY ./tools/www.conf /etc/php/7.3/fpm/pool.d/

RUN chmod +x script.sh

RUN mkdir /run/php/
ENTRYPOINT ["sh", "script.sh"]

CMD ["php-fpm7.3", "-R", "-F"]
```

If you wanna know more about what is WP-CLI (Wordpress Command-line interface) and how to install it click here.

As you make have noticed, there is another script here that I copy inside the container. Here is it's content:

```
cd /var/www/html/wordpress
#Downloading and extracting Wordpress core files to the current directory
wp core download --allow-root

# Creating the wp-config.php file using this command.
wp core config --dbname=${MYSQL_DATABASE} --dbuser=${MYSQL_USER} --dbpass=${MYSQL_PASSWORD} --dbhost=mariadb --allow-root

# Installing wordpress using the given environment variables to avoid showing the wordpress installation page everytime we run the containe
wp core install --url=${DOMAIN_NAME} --title=DopamInception --admin_user=${WP_USER} --admin_password=${WP_PW} --admin_email=${WP_EMAIL} --a

exec $@
```

This script creates a wp-config.php file with the given environment variables, we need a wp-config file to tell Wordpress the settings needed for database connection, and other options that are specific to the site.

Essentially it's like the "brain" of the Wordpress site, providing the necessary information and settings for the site to function properly.

Let's rewind back a little to the Wordpress Dockerfile, you can see php-fpm and seen it mentioned on the subject, so what is is exactly? and why do we need it?

PHP-FPM

FPM is a PHP FastCGI implementation that includes additional features for managing FastCGI processes. Just like FastCGI it creates a pool of processes that can handle incoming requests.

It can be configured to use different process management strategies, such as dynamic or static process allocation. Its commonly used with NGINX web server and typically used in high-traffic web environments where performance and scalibility are important consideration.

Now that we have all services done, we need to run them all together, and in order for us to do that, we need something called **Docker Compose,** so what is it? How and why use it?

Docker Compose

Its a tool for defining and running multi-container Docker applications using a compose file.

The compose file is a YAML file defining services, networks and volumes for a Docker application.

Used to define multi-containers applications.

The compose specification allows one to define a platform-agnostic container based application, such an app is designed as a set of containers which have to both run together with adequate shared resources and communication channels.

Computing components of an application are defined as Services.

Services: an abstract concept implemented on platforms by running the same container image (or configuration) one or more times.

In our case, MariaDB, Wordpress and NGINX are seperate **services.**

Services communicate with each other through Networks.

Networks: is a platform capibility abstraction to establish an IP route between containers whithin services connected together.

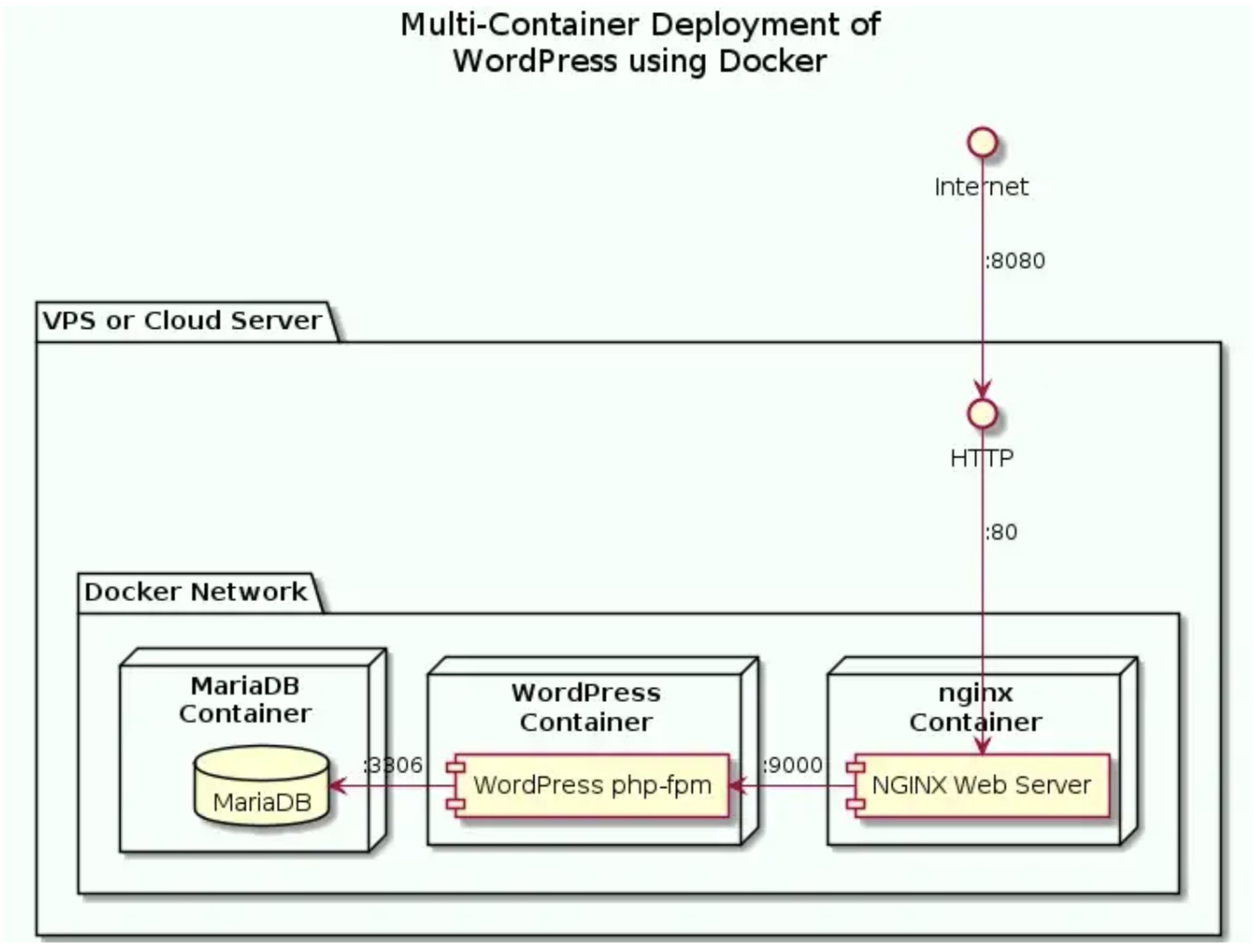
In a less nerdy manner, Networks are like a party conversation circle. It's a space where your guests (containers) can "talk" to each other, share information and work together.

In a nutshell, it allows you to create a secure and isolated environment for your containers to communicate in without exposing them to the outside world.

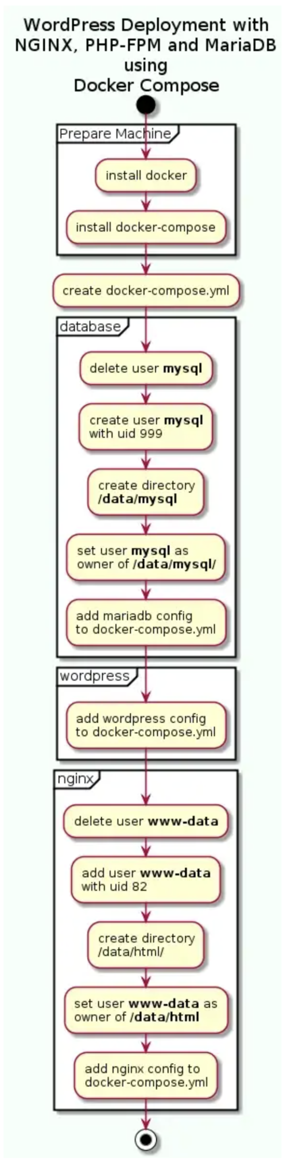
Services store and share persistent data into Volumes.

Volumes: persist data outside the container so it can be backed up or shared. Docker volumes are dependent on Docker's file system. When a container is started, Docker loads the read-only image layer, adds a read-write layer on top of the image stack, and mounts the volumes onto the container filesystem.

Multi-Container deployment of Wordpress using Docker



Overview of the installation process


WordPress Deployment with NGINX, PHP-FPM and MariaDB using Docker Compose
Prepare Machine
install docker
install docker-compose
create docker-compose.yml
database
delete user **mysql**
create user **mysql** with uid 999
create directory **/data/mysql**
set user **mysql** as owner of **/data/mysql**/
add mariadb config to docker-compose.yml
wordpress
add wordpress config to docker-compose.yml
nginx
delete user **www-data**
add user **www-data** with uid 82
create directory /data/html/
set user **www-data** as owner of **/data/html**
add nginx config to docker-compose.yml

Now you may have an idea where we’re going

After creating a docker-compose.yml, we need to setup the services we tackled earlier.

Wordpress configuration

```
wordpress:
    container_name: wordpress
    build: ./requirements/wordpress
    volumes:
      - wordpress:/var/www/html/wordpress
    env_file:
      - .env
    networks:
      - wordpress
    depends_on:
      - mariadb
    environment:
      - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}
      - MYSQL_DATABASE=${MYSQL_DATABASE}
      - MYSQL_USER=${MYSQL_USER}
      - MYSQL_PASSWORD=${MYSQL_PASSWORD}
      - WORDPRESS_DB_HOST=${WORDPRESS_DB_HOST}
      - WORDPRESS_DB_NAME=${WORDPRESS_DB_NAME}
      - WORDPRESS_DB_PASSWORD=${WORDPRESS_DB_PASSWORD}
      - WORDPRESS_TABLE_PREFIX=${WORDPRESS_TABLE_PREFIX}
    ports:
      - 9000:9000
    restart: always
    init : true
```

As you can see we defined the service's name 'wordpress' in the first line.

Then:

- `container_name: wordpress`: As the keyword implies, this line is for naming the container.
- `build`: specifies the directory containing the Dockerfile for building the image for the container.
- `volumes`: sets up a volume named **wordpress** which is mounted to **/var/www/html/wordpress.**

  Since we mentioned **volumes** here why not get on with explaining what they are and why do we use them for.

  ### Volumes

  In Docker a volume is a way to persist data beyond the lifetime of a container. It's essentially a directory or a folder that can be mounted to a container, providing it with acces to the data stored in that directory.

  To put it simply, Volumes are like Dropbox or Google drive. Just like how you create a folder in them to share it with other users, you can create a volume in Docker and share it with other containers.

  The reason why we need them in our case is that we need to share data between the wordpress container and the database container (we'll see that next). By mounting these volumes we ensure that the data generated by the containers, isn't lost when the containers are deleted or restarted.

- `env_file`: specifies the file that contains the environment variables.
- `networks`: creating a network called 'Wordpress' that will be used by the container specified.
- `depends_on`: sets the order on which the services must start and stop, in this case, the mariadb container will run first because the web application (wordpress) needs the database up and running to work properly.
- `environment`: sets the environment variables needed in the container.

NOTE: the Wordpress environment variables:

MYSQL_ROOT_PASSWORD,

WORDPRESS_DB_NAME,

WORDPRESS_DB_USER,

and WORDPRESS_DB_PASSWORD

MUST be matching exactly with the MariaDB environment variables

MYSQL_ROOT_PASSWORD,

MYSQL_DATABASE,

MYSQL_USER,

and MYSQL_PASSWORD.

In case of any mismatch, the **wordpress** service is dependent on the **mysql** service. This implies that the services are started and stopped in the dependency order.

- `ports:` used to map a container's port to a port on the host machine. This allows communication between the container and the host machine or other containers.

  the syntax is as follows `- <host_port>:<container_port>`

- `restart: always`: specifies the restart policy of the container. When its set to always, the docker daemon will restart the container automatically if it exits.
- `init: true`: This tells the docker daemon to run an initialization process before starting the container.

  This is typically used in cases where we need to ensure that certain services or configurations are set up before the container starts such as creating a database schema, or setting environment variables.

NGINX Configuration

adding nginx configuration to the **docker-compose.yml** file as a service named **nginx.**

```
nginx:
    container_name: nginx
    depends_on:
      - wordpress
    build: ./requirements/nginx/
    networks:
      - wordpress
    ports:
      - 443:443
    env_file:
      - .env
    volumes:
      - wordpress:/var/www/html/wordpress
    restart: always
    init : true
```

Similar to the Wordpress service, we set a name for the container, give the needed path to build the image, setup networks and volumes.

And last but not least..

Mariadb Configuration

```
mariadb:
    container_name: mariadb
    build: ./requirements/mariadb/
    restart: always
    environment:
      - MYSQL_ROOT_PASSWOR=${MYSQL_ROOT_PASSWORD}
      - MYSQL_DATABASE=${MYSQL_DATABASE}
      - MYSQL_USER=${MYSQL_USER}
      - MYSQL_PASSWORD=${MYSQL_PASSWORD}
    volumes:
      -  mariadb:/var/lib/mysql/
    env_file:
      - .env
    networks:
      - wordpress
    init : true
```

In this case the only difference is specifying a new volume for mariadb to keep the files easily managed, and well organized.

We're still using the same network because all the services need it to communicate with each other.

Alongside the specification of environment variables needed inside the container.

One more thing…

Since we used networks and volumes we need to define them in sections at the bottom, starting up with networks

```
networks:
  wordpress:
    driver: bridge
```

In the **networks** section we define a network called wordpress with the **driver** set to **bridge** in docker compose this is used to specify the networking driver to be used by the containers in the network. In this case, **bridge** is the default network driver used in Docker.

The **bridge** network driver creates a virtual network that connects with the containers running on the same host. Each container is assigned an IP address and is able to communicate with other containers on the same network using that IP address.

```
volumes :
  wordpress:
    driver: local
    driver_opts:
      type: none
      device: /home/mbaioumy/data/wordpress
      o: bind
  mariadb:
    driver: local
    driver_opts:
      type: none
      device: /home/mbaioumy/data/mariadb
      o: bind
```

The blocks under the **wordpress** and **mariadb** names specify that the volume driver should be **local** which means that the volume is created on the **host machine**.

`driver_opts` provides options for the volume driver, `type` is set to **none** meaning that the volume isn't managed by Docker.

`device` option specifies the path on the host machine where the volume is located.

`o` option specifies the mount option, which in this case is `bind` (when **bind** is used a directory or file on the host is mapped to a directory or file inside the container, which allows data to be shared between the two).

So now our docker-compose.yml file would look something like this..

```
version: "3"

services:
  mariadb:
    container_name: mariadb
    build: ./requirements/mariadb/
    restart: always
    environment:
      - MYSQL_ROOT_PASSWOR=${MYSQL_ROOT_PASSWORD}
      - MYSQL_DATABASE=${MYSQL_DATABASE}
      - MYSQL_USER=${MYSQL_USER}
      - MYSQL_PASSWORD=${MYSQL_PASSWORD}
    volumes:
      -  mariadb:/var/lib/mysql/
    env_file:
      - .env
    networks:
      - wordpress
    init : true
  wordpress:
    container_name: wordpress
    build: ./requirements/wordpress
    volumes:
      - wordpress:/var/www/html/wordpress
    env_file:
      - .env
    networks:
      - wordpress
    depends_on:
      - mariadb
    environment:
      - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}
      - MYSQL_DATABASE=${MYSQL_DATABASE}
      - MYSQL_USER=${MYSQL_USER}
      - MYSQL_PASSWORD=${MYSQL_PASSWORD}
      - WORDPRESS_DB_HOST=${WORDPRESS_DB_HOST}
      - WORDPRESS_DB_NAME=${WORDPRESS_DB_NAME}
      - WORDPRESS_DB_PASSWORD=${WORDPRESS_DB_PASSWORD}
      - WORDPRESS_TABLE_PREFIX=${WORDPRESS_TABLE_PREFIX}
    ports:
      - 9000:9000
    restart: always
    init : true
  nginx:
    container_name: nginx
    depends_on:
      - wordpress
    build: ./requirements/nginx/
    networks:
      - wordpress
    ports:
      - 443:443
    env_file:
      - .env
    volumes:
      - wordpress:/var/www/html/wordpress
    restart: always
    init : true

networks:
  wordpress:
    driver: bridge

volumes :
  wordpress:
    driver: local
    driver_opts:
      type: none
      device: /home/mbaioumy/data/wordpress
      o: bind
  mariadb:
    driver: local
    driver_opts:
      type: none
      device: /home/mbaioumy/data/mariadb
      o: bind
```

Now That we have the docker compose file over with, it’s time to run it!

In order for us to run a docker compose file we need to execute the following command.

```
docker compose up
```

And you’re good to go!

Sources:

https://docs.docker.com/

https://frontendmasters.com/courses/complete-intro-containers/

https://www.plesk.com/blog/various/why-do-you-need-php-fpm/

https://www.digitalocean.com/community/tutorials/php-fpm-nginx

https://blog.devsense.com/2019/php-nginx-docker

https://techviewleo.com/install-wordpress-on-debian-with-nginx-and-lets-encrypt/

https://www.hostinger.com/tutorials/wp-cli

CHATGPT

FastCGI explained