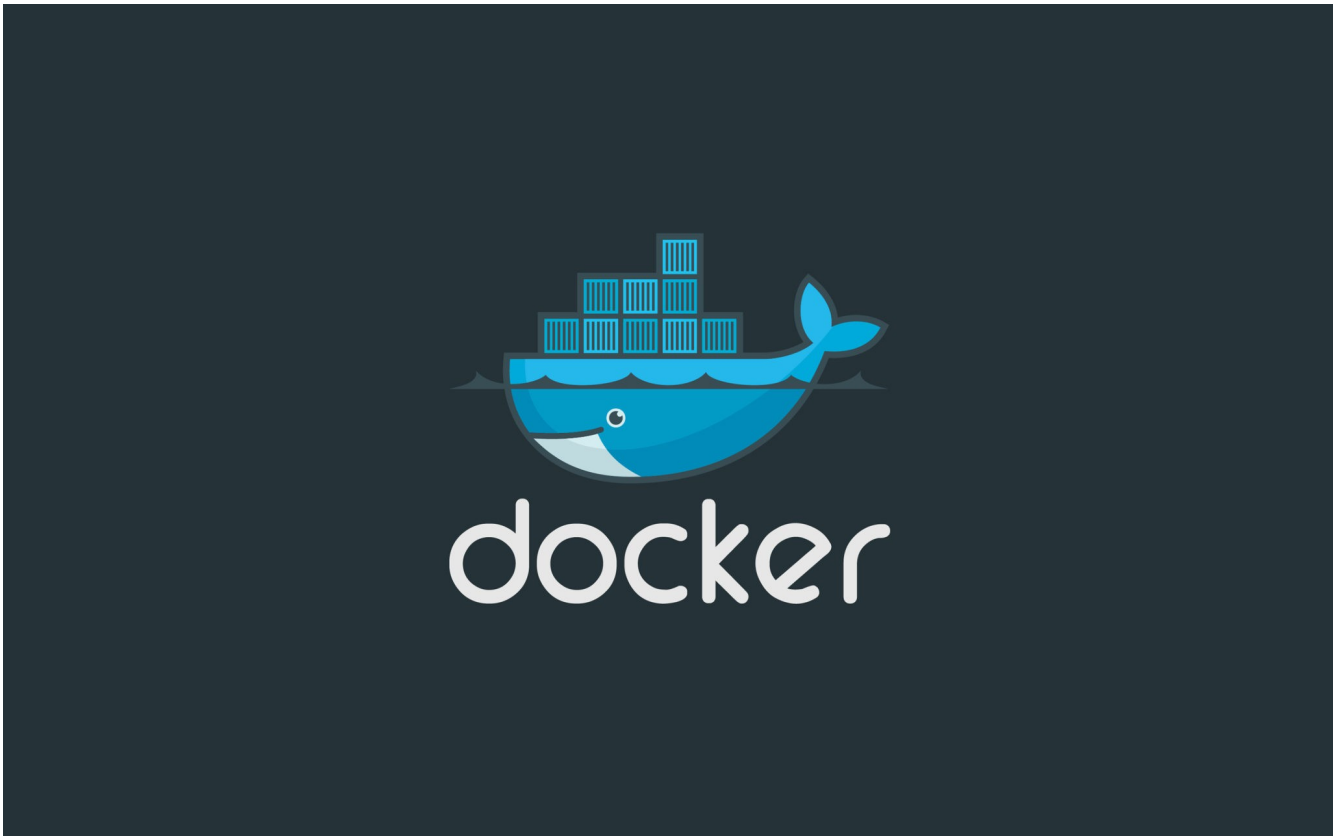

Docker Introduction for development environment with Laravel

1. Basic Commands:
   - *docker image ls* – displays all the available local images to build containers.
   - *docker run {flags}{image:tag} {command}*– creates a docker container based on an image
     - Flags:
       - **--rm** – removes the container once it exits or completes process in foreground.
       - **-i –** interactive mode (used in conjuction with **-t**)
       - **-t –** creates a TTY connection (similar to SSH, but it's not :-]) (use -it for interactive mode)
       - **-d –** runs the container as a daemon (in the background).
       - **-v {location in host} {location in container} –** shares a **volume** (location) from the host machine to the container. It's one time sharing so the volume is not stored!
       - **-p {port on host machine}:{port on container} –** maps the port of the host machine to port in the container.
       - **--name –** gives the container hostname (used in docker networks).
       - **-e –** sets environment variables for the container. Must be set for each ENV variable separately. See docker networks example MySQL
       - **-w {direcotry}** - sets the current working directory (the directory on which the command will be executed – equivalent is **cd {directory}.**
   - *docker stop {container id}* – stops the docker container (loses configuration and stored data).
   - *docker rm {container id}* – removes the container.
   - *docker image ls / docker images {flags}* – Displays all the available local images

        - Flags:
            - **-q –** Displays only the numeric IDs of the images (handy for removing all the images)
            - Application: (**$ docker image rm $(docker images -q)**) **- removes all images**
    - *docker ps {flags}* – shows containers (only running)
        - Flags:
            - **-a –** shows all containers, running and stopped.
    - *docker df {container id}* – shows the changes that have been made to the container in difference with the base image.
    - *docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]* – stores the changes applied to the container within an image (new or already existing one
    - *docker build {OPTIONS} {CONTEXT}* – builds up an image based on an Dockerfile.
        - Options:
            - **-t** – Provide name for the image (tag).
            - **-f –** Location of the Dockerfile
        - Context – Defines the current directory by which Docker will be looking for any file defined within the Dockerfile.
    - *docker exec {flags}* – Runs a command within the containers
        - may be used with **-it bash** to interact with the container.
    - *docker logs {flags}* – displays the logs from stderr and stdout of a running docker container.
        - Flags:
            - **-f –** follows (tails) the logs (keeps updating the log information in real time).
    - *docker network {OPTIONS}* – allows actions with the docker networks
        - Options:
            - **ls –** displays available networks
            - **create {flags} {name} –** creates a new docker network.
            - **inspect {name}** – displays details for the selected network. Used for example to check if a container is part of the network.

2. Docker Images:
    1. In order to build a docker image we will be using ***`Dockerfile`*** as a base from where we can store all the needed configuration in order to create the image. The process is similar as **building it manually from within Docker** and **committing the changes:**
        - For every line within the **Dockerfile** a container is made.
        - Changes are applied to the container.
        - Changes are committed to an image.
        - Container is discarded (except last).
    2. Modification of the images can be made the following way:
        - Grab an Image: **docker run –rm -it ubuntu:18.04 bash**
        - Once within the container, make the needed changes. Ex: *$ apt-get update & apt-get install nginx*.
        - From within another terminal, commit the changes (**or remove –rm flag** to keep the container after exiting): docker commit -a “Your Name Here” -m “Commit message here” container_id(123456789) mynginx:latest
3. Dockerfile
    - TAGS:
        - Label – defines a label for the image
        - ENV – defines an environment variable for the image
        - RUN – executes a command within the image during build

- CMD – run a command.
- ENTRYPOINT – a script that will be executed during the initialization of the container and will fetch the provided argument/s to the run command
- WORKDIR – Location where the all the commands will be run from (Current dir)

```
FROM ubuntu:18.04

LABEL maintainer = "Daniel Yonkov"

ENV DEBIAN_FRONTEND = noninteractive
# setups the datetime in order to avoid some nginx interactions
RUN apt-get update \
&& apt-get install -y gnupg tzdata \
&& echo "UTC" > /etc/timezone \
&& dpkg-reconfigure -f noninteractive tzdata
# builds PHP and Nginx required libraries
RUN apt-get update \
&& apt-get install -y curl zip unzip git supervisor sqlite3 \
  nginx php7.2-fpm php7.2-cli \
  php7.2-pgsql php7.2-sqlite3 php7.2-gd \
  php7.2-curl php7.2-memcached \
  php7.2-imap php7.2-mysql php7.2-mbstring \
  php7.2-xml php7.2-zip php7.2-bcmath php7.2-soap \
  php7.2-intl php7.2-readline php7.2-xdebug \
  php-msgpack php-igbinary \
&& php -r "readfile('http://getcomposer.org/installer');" | php -- --install-dir=/usr/bin/ --
filename=composer \
&& mkdir /run/php \
&& apt-get -y autoremove \
&& apt-get clean \
&& rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
```

- Example docker file
- **Recommendations – LEAST possible RUN instructions**

4. Serving Web Files

We need to setup Nginx in order to handle Requests and send them to PHP-FPM.
In order to do that, we need to add a configuration within our docker container. We do that by ADDing a configuration file for Nginx during the docker image build.

ADD default /etc/nginx/sites-available/default

Contents of ***default***:

```
server {
    listen 80 default_server;

    root /var/www/html/public;

    index index.html index.htm index.php;

    server_name _;

    charset utf-8;

    location = /favicon.ico { log_not_found off; access_log off; }
```

```
    location = /robots.txt  { log_not_found off; access_log off; }

    location / {
        try_files $uri $uri/ /index.php$is_args$args;
    }

    location ~ \.php$ {
        include snippets/fastcgi-php.conf;
        fastcgi_pass unix:/run/php/php7.2-fpm.sock;
    }

    error_page 404 /index.php;
}
```

Later on we create a testable files that we can share with the root defined within Nginx (/var/www/html/public)

Let's create a new folder where we will contain the web app within our host machine:

- make a folder with name of your choice (I will be naming it application)
- create the index.php file to show phpinfo();
- create the index.html file to show some dummy data (Hello, Docker)

We can see that currently only the Index.html file is served, as we haven't yet handled the PHP-FPM process to be running on the foreground.

5. Running multiple processes:
    - In order to handle PHP files we need a middle man (gateway) to communicate between Nginx and PHP, we need PHP-FPM!
    - We can run though only one process during the startup of the container, so we will use a service called **Supervisor** to handle spinning multiple processes in the foreground.
        - First we need the configuration for Supervisor, in order to know which services to run:
            - Let's create the file within docker/app/**supervisord.conf**

```
[supervisord]
nodaemon=true

[program:nginx]
command=nginx
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0

[program:php-fpm]
command=php-fpm7.2
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0
```

            - We add the configuration to supervisor from within our Dockerfile
              ADD supervisord.conf /etc/supervisor/conf.d/supervisord.conf
            - Let's add to run as default command supervisor if no command is provided.

CMD [“supervisord”]

6. Run PHP-FPM as a daemon

This is required in order to get it to run with **supervisor** and the provided configuration.

- Edin **/etc/php/7.2{your version here}/fpm/php-fpm.conf** > find **daemonize=yes** and change it to “**no**”
- Copy the configuration within a new file by which we can build the docker image (php-fpm.conf within docker/app/)
- add the configuration file within the **Dockerfile**
  ADD php-fpm.conf /etc/php/7.2/fpm/php-fpm.conf

7. Docker Logs
   Anything that Docker runs in the foreground process, he monitors within the stderr and stdout (dev/stdout ; dev/stderr).
   Since we are handling **PHP-FPM** and **Nginx** through **supervisor** , if we wanted to see the output of the logs from within Nginx and PHP-FPM, we will need to get their input to dev/stderr and dev/stdout.
   1. Process:
      - create simlinks to error log files within the docker container for PHP-FPM and Nginx
      - For PHP-FPM you can alternatively change logs to store within `/proc/self/fd/2`
        **RUN ln -sf /dev/stdout /var/log/nginx/access.log **
        **&& ln -sf /dev/stderr /var/log/nginx/error.log **
        && ln -sf /dev/stderr /var/log/php7.2-fpm.log

8. Entrypoint – defines a command which will always be run on initialization of the container without regarding if any command has been given. If a command is given, it is passed to the Entrypoint script. **This script must be accessible from within the docker container!**
   Due to some bug with composer directory on native Linux users, we may need to add the following Entrypoint script

```
#!/usr/bin/env bash

##
# Ensure /.composer exists and is writable
#
if [ ! -d /.composer ]; then
    mkdir /.composer
fi

chmod -R ugo+rw /.composer

##
# Run a command or start supervisord
#
if [ $# -gt 0 ];then
    # If we passed a command, run it
    exec "$@"
else
    # Otherwise start supervisord
    /usr/bin/supervisord
fi
```

9. Docker Networks
   Networks are used in order to allow communication between containers.
   For this task we will be using a MySQL container and linking it with our App container (will use default Laravel configurations in this example):

```
docker run --rm -d \
--name=mysql \
--network=appnet \
-e MYSQL_ROOT_PASSWORD=root \
-e MYSQL_DATABASE=homestead \
-e MYSQL_USER=homestead \
-e MYSQL_USER_PASSWORD=secret \
Mysql:5.7
```

```
docker run --rm -d \
    --name=app \
    --network=appnet\
     shippingdocker/app:latest
```

10. Connecting containers.
    Since we have added those containers in the same network – they have been connected already. The only thing we have to keep in mind is the name we have given to the containers in order to use them as host names for the configuration required by our web app. For example, instead of connecting to localhost to MySQL, we will be connecting to the container named “mysql”, so we will be using address = mysql.