Docker Introduction for development environment with Laravel part 2 (Docker Compose)

1. Docker Compose – docker-compose is an API behind docker which makes easy handling the structure for the docker containers, networks, volumes etc. With much shorter and interactive commands.

As you can see in the repository commands for the pure docker environment, it's commands are long and some of them require to be done separately in order to persist data and connect the different containers to each other (**docker volumes** and **docker networks**) this is the reason we are going to explore the *docker-compose* library, in order to display a different approach.

2. docker-compose.yml – In order to configure our docker-compose environment, we need a **docker-compose.yml** file, which will serve us in a similar way as our *Dockerfile*, (building the environment and setting it up).
   - **Structure of docker-compose.yml:**
     - **version –** Defines the explicit version of docker-compose that we will be using, if omitted will use the latest version (probably)
     - **services –** The docker containers that we want to use (Ex: Nginx+PHP {app container} MySQL, Redis etc...)
     - **networks –** Definition of the network/s that docker-compose will create and we will use to connect our docker containers.
     - **volumes -** Similar to networks, the definition of volumes that docker-compose must create and that we will be using in our persisting data storage containers, in order to not lose data when we destroy our containers.

            - Example File:

```
version: '3'
services:

networks:
  appnet:
    driver: bridge
volumes:
  dbdata:
    driver: local
  cashdata:
    driver: local
```

3. docker-compose commands
    - **docker-compose ps -** Similar to *docker ps* – it is specific to only containers running under docker-compose.
    - **docker-compose up {flags}** – similar to docker run + docker build if is provided an image based on *Dockerfile*.
        - -d – Daemon {run in the background}.
    - **docker-compose down –** stops and destroys the containers.
    - **docker-compose stop** - stops but is doesn't destroy the containers
    - **docker-compose start –** starts the stopped containers.
    - **docker-compose exec** - similar to docker exec, does not support setup of working dir.
4. Docker Services:
   We define our containers within the **Services** container within the *docker-compose.yml* file. First, we will define the images that doesn't require modification from our part and in this case they are **Redis** and **MySQL.**
   Example for the service segment:

```
services:
  # naming based on use case, not of technology
  cache:
    image: redis:alpine
    networks:
      - appnet
  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: homestead
      MYSQL_USER: homestead
      MYSQL_PASSWORD: secret
    networks:
      - appnet
```

    - Explanation of the tree structure:
        - first level – Service name, this will be the hostname of the service we want to access, in those examples **"cache"** and **"db"** reporesent Redis and MySQL
        - Second level:

        - **Image:** Defines the image that we will be using for this service, same as the one we run under *docker run* or the one that we base on *Dockerfile(FROM {image})*
        - **networks:** Defines to which networks this container will be connected, allowing to communicate with the containers in that network.
        - **environment:** Defines ENVIRONMENT VARIABLES available for the build process of any service that may require them.

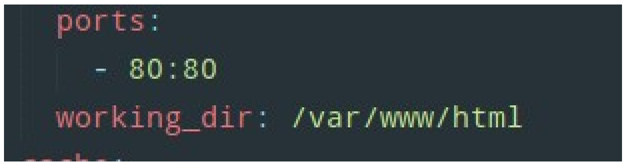
5. Binding image volumes to persistant volumes from docker-compose.
    - In order to persist the data, since the images do generate usually their own volumes, we need to link the persistant docker volumes that we created to the ones that are generated by the images.
      This information can be found in hub.docker.com – finding the image tag and seeing within the *Dockerfile* of that image, where the volume is pointing to.
6. Building the App service:
   Since our PHP + Nginx image is a custom build image not uploaded to hub.docker.com, we will need to **Build** that image if it's not available in our docker images list.
    - Example of the building the **docker-compose.yml** file for this **service:**

```
app:
  build:
    context: ./docker/app
    dockerfile: ./docker/app/Dockerfile
  image: shippingdocker/app:latest
  networks:
    - appnet
  volumes:
    - ./application:/var/www/html
  ports:
    - 80:80
```

    - Similar to the previously build of the services, we will look further into some additional details for the build of this image if it's lacking on our docker images list and how to do it with our custom build *Dockerfile*
        - **Build** – defines what to do if the image is not found locally or in hub.docker.com.
            - **Context –** similar to the docker build command, we define from which directory we will be looking for the files mentioned within the *Dockerfile* (current directory to execute from the Dockerfiel)
            - **Dockerfile –** The location of the *Dockerfile* from which we will build the custom image.
        - **Image –** the name of the image that we will pull from docker-hub, if available, but in this context, it will be considered as the custom image that we build.
        - **Ports:** Defines the port that we map from {host machine}:{docker container}.

7. Setup working directory in **docker-compose.yml**
   Currently the version of docker-compose it's used in this documentation is 1.22 – but in future versions, **this may work differently.**
   docker-compose exec does not take into consideration the working directory provided within the **docker-compose.yml** so we need to use a trick for this:
   docker-compose exec ***bash -c "CD into the working dir && your command in the context of the directory"*** Keep in mind that this is a string, so it needs to be wrapped in a quote marks.

- In order to setup the working dir, we use the keyword **working_dir** and we apply it to our **app** service like so:



```
ports:
  - 80:80
working_dir: /var/www/html
```

8. Docker environment variables:
   Allows setting up more dynamically your environment via variables set within the bash/shell environment or an even better option is to have a **.env** file containing the variable name and value to be used under **docker-compose.yml**
   - docker-compose syntax for the variable within docker-compose.yml:
     - The syntax is as follows: $**{VARIABLE_NAME}**
   - We can also archive this effect by exporting a LOCAL VARIABLES for the current open terminal:
     ```
     export DB_PORT=33060
     export APP_PORT=8080
     docker-compose up
     ```
   - Or even providing it within the scope of *docker-compose:*
     ```
     APP_PORT=8080 DB_PORT=33060 docker-compose up -d
     ```
   - Or a **.env** within the same directory as our **docker-compose.yml** configuration file - really suitable for Laravel dev environment since it already uses .env as it's primary configuration file:
     .env file content:
     ```
     APP_PORT=8080
     DB_PORT=33060
     ```

We can now connect our SQL client to our docker DB service.

9. NodeJS service:
   In order to compile our Laravel assets – we may want to use NodeJS, since this is not a constantly running service, but instead is used when needed from our side. We can simply initialize it without running anything in the foreground which will make the container EXIT the monitoring and stop the container. WE will use docker run or docker-compose run to run the commands we wish to execute at certain time.
   Example of NodeJS *Dockerfile* (since we need to add some modifications to it)

```
FROM node:latest

LABEL maintainer="Daniel Yonkov"

RUN curl -sS https://dl.yarnpkg.com/debian/
pubkey.gpg | apt-key add - \
    && echo "deb
http://dl.yarnpkg.com/debian/ stable main" >
/etc/apt/sources.list.d/yarn.list \
    && apt-get update \
    && apt-get install -y git yarn \
    && apt-get -y autoremove \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/* /tmp/*
/var/tmp/*
```

Configurations within docker-compose.yml

```
node:
    build:
      context: ./docker/node
      dockerfile: Dockerfile
    image: shippingdocker/node:latest
    networks:
     - appnet
    volumes:
      - ./application:/opt
    working_dir: /opt
```

And let’s build or node module as follows:
docker-compose run --rm node yarn nstall.

10. Workflow:
 Since we need to run commands on our server and the process of running commands maybe tedious for some simple actions like migrating tables or creating models in Laravel, we can make our docker commands much simpler with a simple executable script.
    - Create a named file by your choice (I will be using develop as it’s relates to the development evironment)
    - Define which bash/shell to use (first line of the file): *#!/usr/bin/env bash*
    - Docker + Laravel + Composer commands:
        - Ok we need the following commands to work properly with Laravel, composer and maybe some docker-compose,as it would be easier not to type out the hifen
            - docker-compose up/down

```
#!/usr/bin/env bash
if [ $# -gt 0 ]; then
        if [ "$1" == "start" ]; then
                docker-compose up -d
        elif [ "$1" == "stop" ]; then
                docker-compose down
        fi
else
        docker-compose ps
fi
```

Since this a bash scripting language, let’s define some of the special character combinations:
- **$#** - number of arguments provided to the executable file (command)
- **-gt** – greater than
- **if [“$1” == “start”]; then** – if statement.

        - **$1 –** first argument provided
        - **then –** upon successfully matching the if condition, what action needs to be taken.
    - **Elif [.….]** - elseif statement (we use these instead of a switch statement)
    - **else –** else statement, when nothing else matches
    - **fi –** end of if statement

  Ok so what is that we do here? We check if we have provided any arguments to the executable script. If none are provided we simply run **docker-compose ps,** else we check if the first argument provided is **"start"** or **"stop"** and we if there is a match, we run either **docker-compose up -d** or **docker-compose down.**

- Artisan:
  Since artisan uses multiple params provided to it, we need to edit a bit the arguments provided to the artisan command. Let's see how we do it:

```
elif [ "$1" == "artisan" ] || [ "$1" == "art" ]; then
        shift 1
        docker-compose exec app \
        php artisan "$@"
```

  Ok we check if the command provided is either **"artisan"** or **"art"** and if so, we do a **shift 1** (this command moves the provided parameters with 1, which removes the first one provided, but keeps all the rest in memory) and we execute the **php artisan** command with the rest of the parameters provided for the artisan command (**$@** - paste all params, flags etc. provided).

- Composer
  Similar to Artisan, we do only need to change the last command:

```
elif [ "$1" == "composer" ] || [ "$1" == "comp"];
then
        shift 1
        docker-compose exec app \
        composer "$@"
```

- Unit tests:
  in order to run the unit tests for Laravel app:

```
elif [ "$1" = "test" ]; then
        shift 1
        docker-compose exec app \
        ./vendor/bin/phpunit "$@"
```

- NodeJS commands:
  Also if no command between the ones provided is matched, we run **docker-compose** *command* :)

```
elif [ "$1" == "npm" ]; then
        shift 1
        docker-compose run --rm \
            node \
            npm "$@"

    elif [ "$1" == "yarn" ]; then
        shift 1
        docker-compose run --rm \
            node \
            yarn "$@"

    elif [ "$1" == "gulp" ]; then
        shift 1
        docker-compose run --rm \
            node \
            ./node_modules/.bin/gulp "$@"
    else
        docker-compose "$@"
```