Что такое Docker-Compose?

Docker-compose это как дирижёр оркестра, где ваш оркестр - это набор контейнеров, которыми нужно управлять. Каждый контейнер имеет отдельную задачу, как и музыкальные инструменты в разных частях песни.
Docker Compose управляет контейнерами, запускает их вместе, в нужной последовательности, необходимой для вашего приложения.
Его можно назвать дирижёром в мире Docker-а.
Docker-compose организовывает совместных запуск контейнеров, как инструменты в групповой игре в определённых участках песни.

Каждый инструмент имеет конкретную задачу в группе оркестра. Труба создаёт основную мелодию песни; фортепиано, гитара дополняют основную мелодию; барабаны задают ритм; саксофоны добавляют больше гармонии и т.д.
Каждый из инструментов имеет свою конкретную работу и задачу, как и наши контейнеры.

Docker-compose написан в формате YAML который по своей сути похож на JSON или XML. Но YAML имеет более удобный формат для его чтения, чем вышеперечисленные. В формате YAML имеют значения пробелы и табуляции, именно пробелами отделяются названия параметров от их значений.

Создадим новый файл `docker-compose.yml`, для рассмотрения синтаксиса Docker Compose:

```yaml
version: '3'

services:
  app:
    build:
      context: .
    ports:
      - 8080:80
```

Теперь, построчно разберёмся с заданными параметрами, и что они значат:
version: какая версия docker-compose используется (3 версия - самая последняя на даный момент).
services: контейнеры которые мы хотим запустить.
app: имя сервиса, может быть любым, но желательно, чтобы оно описывало суть этого контейнера.
build: шаги, описывающие процесс билдинга.
context: где находится Dockerfile, из которого будем билдить образ для контейнера.
ports: маппинг портов основной ОС к контейнеру.
Мы можем использовать этот файл для билдинга нашего предыдущего образа apache:

```
docker-compose build
```

После выполнения этой команды, Docker спарсит файл docker-compose и создаст описанные сервисы на основе инструкций во вкладке **build**.
А **context** говорит о том, из какой директории мы берём Dockerfile для создания образа сервиса (в текущем случае - это означает текущую директорию `.`, но могло быть и `/php-cli`, `/nginx`, и т.д.).
И теперь, запустим эти сервисы, которые создали:

```
docker-compose up
```

В результате чего, сервер должен был запуститься, и стать доступным по адресу localhost:8080.
Теперь, отключитесь от консоли, нажав CTRL+C.

Когда контейнер под названием app запускается, docker-compose автоматически связывает указанные порты во вкладке `ports`. Вместо того, как мы делали ранее,

выполняя `-v 8080:80`, docker-compose делает это за нас, получив информацию параметра `ports`. Docker-compose избавляет нас боли, связанной с указанием параметров в командной строке напрямую.
С docker-compose.yml мы переносим все параметры, ранее записываемые в командной строке при запуске контейнера в конфигурационный YAML файл.
В этом примере мы записали BUILD и RUN шаги для нашего сервиса в *docker-compose.yml*. И преимущество такого подхода ещё в том, что теперь для билдинга и для запуска этих сервисов нам нужно запомнить только 2 команды: `docker-compose build` и `docker-compose up`. При таком подходе не нужно помнить, какие аргументы нужно указывать, какие опции нужно задавать при запуске контейнера.
Теперь давайте сделаем нашу разработку немного легче. Сделаем, чтобы вместо постоянного перестроения образа при каждом изменении файлов, мы можем примонтировать нашу рабочую папку в контейнер.
Для этого, удалите строку `COPY . /var/www/html` с Dockerfile - теперь все файлы будут прокинуты из основной ОС. Новый Dockerfile будет иметь вид:

```
FROM php:7.2-apache
WORKDIR /var/www/html
RUN apt-get update && apt-get install -y wget
EXPOSE 80
```

Ранее мы рассматривали, как примонтировать папку в контейнер, для этого мы запускали контейнер с аргументом `-v <HOST_DIRECTORY>:<CONTAINER_DIRECTORY>`.
С Docker-compose мы можем указать напрямую в *docker-compose.yml*:

```
version: '3'
services:
  app:
    build:
      context: .
    ports:
      - 8080:80
```

```
    volumes:
      - .:/var/www/html
```

Добавленная строка примонтирует текущую директорию основой операционной системы к директории /var/www/html контейнера.

В отличие от указания путь в консоли, здесь можно указывать относительный путь (`.`), не обязательно указывать полный путь (`C:\projects\docker-example\apache`), как было ранее при ручном запуске контейнера.
Теперь, выполните по очереди команды:

```
docker-compose stop
docker-compose rm
```

При удалении, вас спросят, действительно ли удалять, напишите `y` и нажмите кнопку enter. Эти команды остановят и удалят все контейнеры, описанные в файле *docker-compose.yml* (то же самое, как мы ранее запускали `docker stop <CONTAINER_ID>` и `docker rm <CONTAINER_ID>`)
Теперь перебилдим сервисы, потому что мы изменили Dockerfile:

```
docker-compose build
```

И заново запустим:

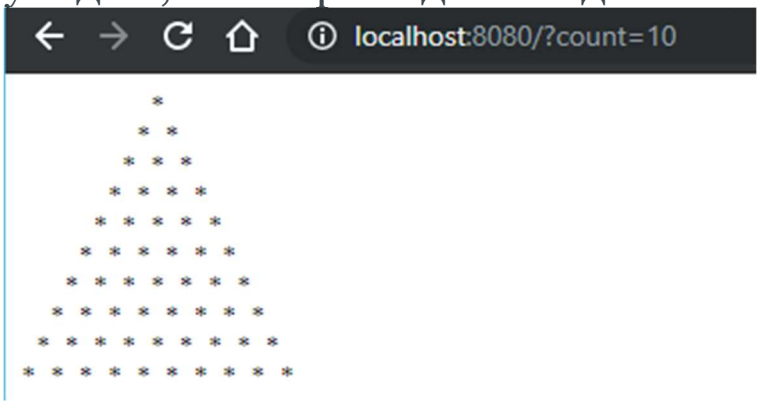
```
docker-compose up
```

И опять, по адресу localhost:8080 поднимется наш сервер. Вместо того, чтобы копировать каждый раз файлы в образ, мы просто примонтировали папку, содержащую исходный код приложения. И теперь, каждый раз не придётся делать ребилд образа, когда файлы изменяются, теперь изменения происходят в лайв режиме, и будут доступны без перестройки образа.

Чтобы в этом убедиться, изменим файл *index.php*, добавим в него скрипт нами любимой пирамиды:

```php
<?php
$n = $i = $_GET['count'] ?? 4;
echo '<pre>';
while ($i--) {
   echo str_repeat(' ', $i).str_repeat('* ', $n - $i)."\n";
}
echo '</pre>';
```

И теперь, если перейти по адресу localhost:8080?count=10, то увидим, что пирамида выводится:



Монтирование вашей локальной папки как Docker Volume это основной метод как разрабатывать приложения в контейнере.

Так же, как мы ранее выполняли команды внутри контейнера, указывая аргументы `-it ... /bin/bash`. Docker-compose так же предоставляет интерфейс по удобному выполнению команд внутри конкретного контейнера. Для этого нужно выполнить команду:

```
docker-compose exec {CONTAINER_NAME} {COMMAND}
```

где, вместо `{CONTAINER_NAME}` нужно записать имя контейнера, под которым он записан в сервисах;
а вместо `{COMMAND}` - желаемую команду.
К примеру, эта команда может выглядеть так:

```
docker-compose exec php-cli php -v
```

Но, сделаем это на основе текущего Dockerfile:

```
docker-compose down  # остановим контейнеры
docker-compose up -d  # здесь используется опция -d которая сообщает, что контейнер должен висеть в режиме демона
docker-compose exec app apache2 -v
```

Пока что, в docker-compose.yml описан только один сервис, потому разворачиваем мы только один контейнер. Но реальный файл *docker-compose* выглядит больше. К примеру, для Laravel он такой:

```
version: '3'
services:
  nginx:
    build:
      context: ./
      dockerfile: docker/nginx.docker
    volumes:
      - ./:/var/www
    ports:
      - "8080:80"
    depends_on:
      - php-fpm
  php-fpm:
    build:
      context: ./
      dockerfile: docker/php-fpm.docker
    volumes:
      - ./:/var/www
    depends_on:
      - mysql
      - redis
    environment:
      - "DB_PORT=3306"
      - "DB_HOST=mysql"
```

```
      - "REDIS_PORT=6379"
      - "REDIS_HOST=redis"
  php-cli:
    build:
      context: ./
      dockerfile: docker/php-cli.docker
    volumes:
      - ./:/var/www
    depends_on:
      - mysql
      - redis
    environment:
      - "DB_PORT=3306"
      - "DB_HOST=mysql"
      - "REDIS_PORT=6379"
      - "REDIS_HOST=redis"
    tty: true
  mysql:
    image: mysql:5.7
    volumes:
      - ./storage/docker/mysql:/var/lib/mysql
    environment:
      - "MYSQL_ROOT_PASSWORD=secret"
      - "MYSQL_USER=app"
      - "MYSQL_PASSWORD=secret"
      - "MYSQL_DATABASE=app"
    ports:
      - "33061:3306"
  redis:
    image: redis:3.0
    ports:
      - "6379:6379"
```

И при выполнении одной только команды `docker-compose up`, поднимутся 5 сервисов. В сравнении с тем, что мы бы вручную выполняли 5 раз команду `docker run ...`. Так что, использование docker-compose в этом случае - очевидно. Так что, теперь, ещё один шаг позадчи, теперь вы знаете, **что такое docker и docker compose**, и для чего нужен каждый из них.