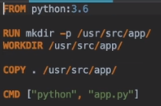
DOCKER, DOCKER COMPOSE

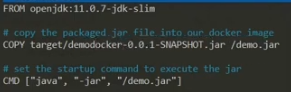
DOCKER

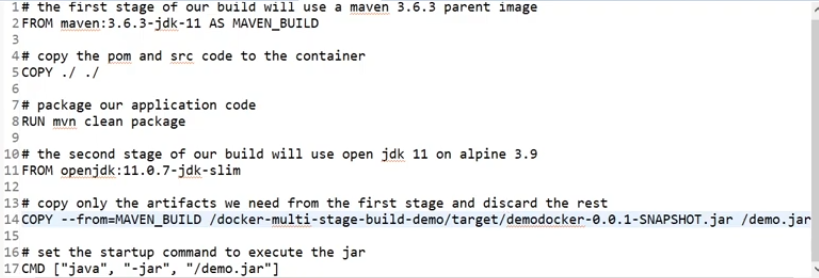
Создать в корневой папке проекта файл “Dockerfile”, в котором прописать нашу сборку:

видеоролик - https://www.youtube.com/watch?v=QF4ZF857m44:



видеоролик - https://www.youtube.com/watch?v=ra80QpzrAWk:





где:

**FROM** – базовый образ (уже готовый и лежащий в docker-хаб) с которого мы начинаем сборку. Тут до двоеточия идет название базового образа, а после его версия(“тэг”)*(если тэг не напишем, то по умолчанию будет написано “latest” )*.

**RUN** – говорит, что нужно выполнить написанную на против него команду. Тут создать папку.

**WORKDIR** – создается рабочая директория внутри образа и переход в нее. И именно в ней, а не в корневой папке (“/”), нужно размещать все файлы приложения.

**COPY** = может принимать а). два или б). три параметра: а). *1.* откуда копировать (у нас “.” с текущей директории*(с нашей машины)*) *2.* куда *(в контейнер)* копировать б). *1.* из какого из FROM копировать *2.* откуда копировать (у нас “.” с текущей директории*(с нашей машины)*) *3.* куда *(в контейнер)* копировать

**CMD** – что нужно докеру делать, когда мы запустим контейнер командой “docker run” или “docker start”, у нас тут говорится что нужно запустить “python” и он выполнит файл “app.py”(а этот файл после команды **COPY** должен будет уже находиться в нашей рабочей директории, созданной командой **WORKDIR,**).

**EXPOSE** 8080 – мы говорим выводить из контейнера информацию на порт 8080,

**ENV** TZ Europe/Kiev - переменная окружения с указанным моим часовым поясом, или можно переменную окружения указывать при написании команды “docker run \*\*\* –e TZ=Europe/Kiev \*\*\*”

**Dockerfile** – (я так понял, что) это просто файл без указания типа файла.

Общие команды:

Ctrl+c - выходим из работающего не в фоне контейнера;

cd C:\papka\podpapka – переход в нужную папку;

cd ../FFF – поднимаюсь на уровень выше и там захожу в папку FFF;

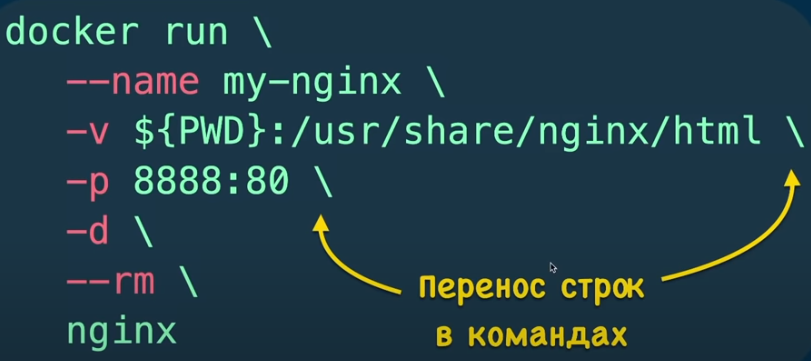
nano Dockerfile – создаем файл с названием “Dockerfile”, при этом у автора ролика этот файл еще сразу открылся в его командере (Cmder) для редактирования;

ls – покажет внутренние папки в средах типа “powershell”;

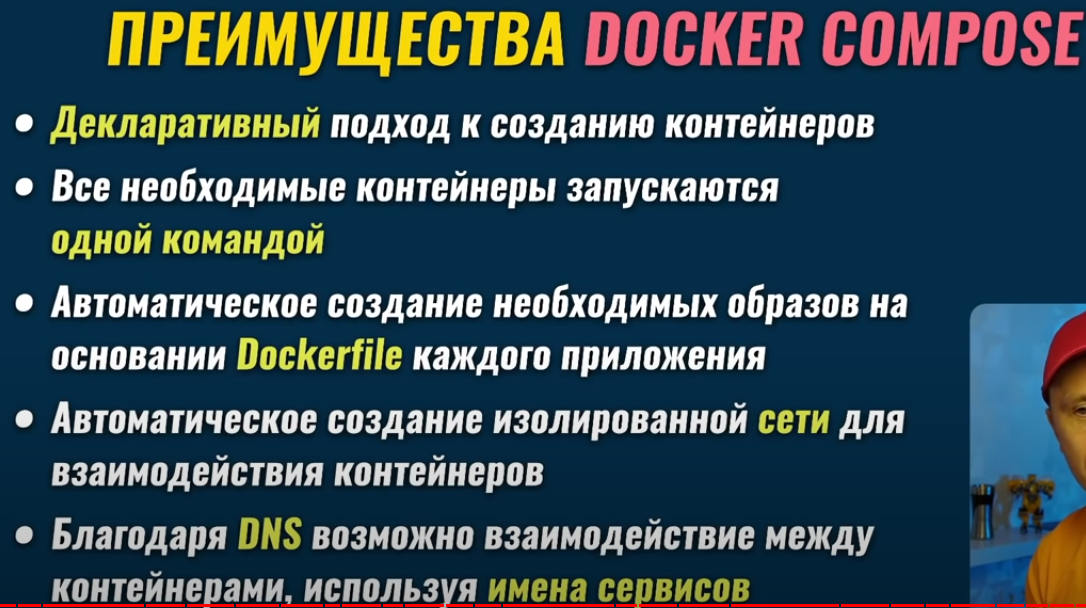
Запускать следующую команду нужно находясь в корневой папке своего проекта, а не ниже:

mvn clean package - упаковка в JAR(по умолчанию)/WAR/EAR;

Команды:

* docker **build -t имя\_создаваемого\_Образа:наш\_тэг** **.***(если мы находимся в папке где лежит файл “Dockerfile”)* – создаем новый образ с указанным нами именем и тэг-ом *(если тэг не напишем, то по умолчанию будет написано “latest” )*, где будет использована информация по сборке из файла “Dockerfile”. Если изменить что-то в файле “Dockerfile” и опять набрать эту-же команду, то образ создастся с тем же именем, а старый образ получит имя <none> с тэг-ом <none>.;
* docker **build** -t имя\_создаваемого\_Образа . **-f название\_докерфайла** – указывается если название нашего докерфайла отличается от “Dockerfile”;
* docker **images** – покажет все образы которые у нас есть;
* docker **ps** – покажет работающий сейчас контейнер;
* docker **ps -a** - покажет все контейнеры, включая отработавшие;
* docker **run имя\_запускаемого\_Образа** – запускает выбранный нами образ с тэг-ом по умолчанию “latest”, и также запустит контейнер в котором выполниться наш файл;
* docker **run** имя\_запускаемого\_Образа**:тэг** – запускает выбранный нами образ с выбраным нами тэгом, и также запустит контейнер в котором выполниться наш файл;
* docker **run --name hello** имя\_запускаемого\_Образа – тут мы говорим, что нужно запустить образ имя\_запускаемого\_образа и при этом создать контейнер не с произвольным именем, а указанным нами “hello”;
* docker **run** --name hello **-d** имя\_запускаемого\_Образа - -//-, но при этом контейнер запуститься и будет работать в фоне и не будет нам мешать запускать другие команды в терминале;
* docker **run** --name hello -d **--rm** имя\_запускаемого\_Образа - -//-, при этом контейнер сам удалиться из списка либо после окончания работы программы в нем работающей либо после принудительной остановки контейнера;
* docker **run** --name hello -d --rm **-p 8080:8080 (или 8080:80 или еще как угодно)**  имя\_запускаемого\_Образа - выводим ip-соединение наружу контейнера, где 1-й порт это порт нашего компьютера где запускается докер контейнер(так называемый “внешний порт”) который будет видно остальным микросервисам/пользователям снаружи в их браузерах и на который они смогут заходить, 2-й порт это внутренний порт запущенный внутри самого докер контейнера. При чем одновременно можно открыть 2 или больше одинаковых контейнеров но с разными внешними портами (8081, 8082 и т.д.);
* docker **run** --name hello -d --rm -p 8080:8080 **-e TZ=Europe/Kiev** имя\_запускаемого\_Образа - -//-, переменная окружения с часовым поясом, или можно через написания переменной в “Dockerfile”;
* docker **run** --name hello -d --rm -p 8080:8080 -e TZ=Europe/Kiev **-** **v абсолютный\_путь\_на\_хостовой\_машине*(или ${PWD} - где ${PWD} это переменная обозначающая абсолютный путь к текущей папке)* :абсолютный\_путь\_внутри\_докер\_контейнера** имя\_запускаемого\_Образа - -//-, для взаимодействия контейнера с БД прикручиваем папку из нашего проекта, лежащую на компе, к папке в контейнере(у нас на рисунке это /usr/src/app/ + тоже название папки, которой заканчивается абсолютный путь на хостовой машине), и получаем возможность изменять данные получаемые/обрабатываемые/выводимые при работе контейнером без монтирования нового образа после каждого изменения каких-то данных в файлах в этой папке + после окончания работы контейнера сохраняться изменения внесенные в файлы такой папки ;
* docker **rm***(без 2-х черточек перед ним, как было выше)* **имя\_ Контейнера –** удаляем конкретный контейнер;
* docker **container prune** – удаление всех остановленных контейнеров;
* docker **rm $(docker ps -qa)** – для удаления всех контейнеров;
* docker **rmi имя\_имиджа** – удаление имиджа;
* docker **rmi $(docker images -q)** – удаление всех имиджей;
* docker **stop имя\_работающего\_Контейнера** – принудительно останавливаем работающий контейнер;
* docker **volume ls** – показывает наличие volum-ов (папок-надстроек над контейнером, которые значительно чаще используются чем их аналог “docker run \*\*\*\* -v \*\*:\*\* \*\*\*”);
* docker **volume create название\_volume\_папки** – создаем volume-папку с таким названием;
* docker **run** --name hello -d --rm -p 8080:8080 -e TZ=Europe/Kiev **-** **v название\_volume\_папки:абсолютный\_путь\_внутри\_докер\_контейнера** имя\_запускаемого\_Образа – так же используется для взаимодействия контейнера с БД;
* docker run **–it** имя\_запускаемого\_Образа – мы заходим с помощью подгруженной у нас оболочки (для образа “openjdk” это jshell) внутрь контейнера загруженного из этого образа. И тогда я могу вводить внутренние (для этой оболочки) команды;
* docker logs **имя\_работающего/остановленного\_Контейнера** – если мы хотим посмотреть логи какого-то конкретного контейнера;
* docker **container inspect** имя\_работающего\_Контейнера – покажет всю информацию о работающем контейнере (тут в секции “Mounts”: после “Source:” будет написан путь к привязанной папке на нашем компьютере, в после “Destination:” будет написан путь к папке внутри контейнера;
* docker **container inspect** имя\_работающего\_Контейнера **|**(вертикальная черточка) **grep IPAdress –** покажет только ipadress работающего контейнера;
* docker **exec** **какие-то\_до.параметры\_запуска**(типа –it или др.) имя\_работающего\_Контейнера **имя\_процесса\_который\_будет\_запущен\_внутри\_работающенго\_контейнера** – дополнительно выполнит какой-то процесс/команду в уже работающем контейнере;
* для лучшей читабельности длинной команды ее можно записать следующим образом: 

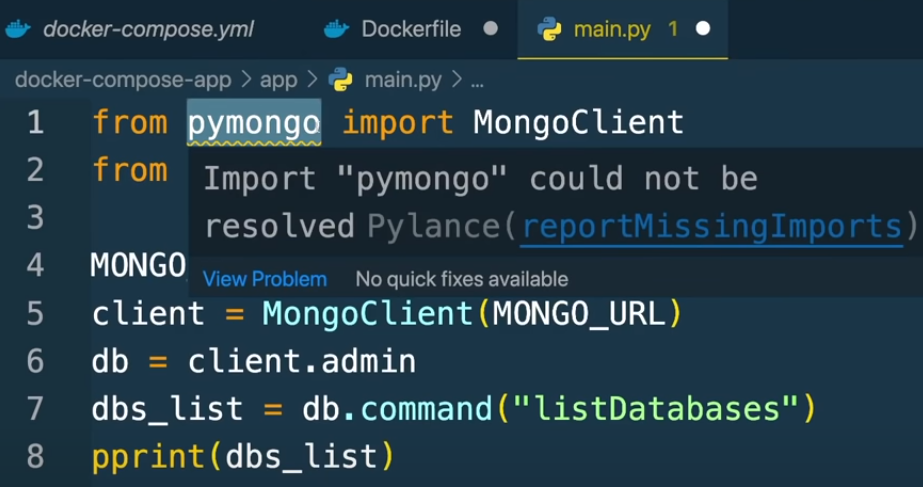
DOCKER COMPOSE



Создать в корневой папке проекта файл “docke-compose.yml”, в котором прописать нашу сборку:

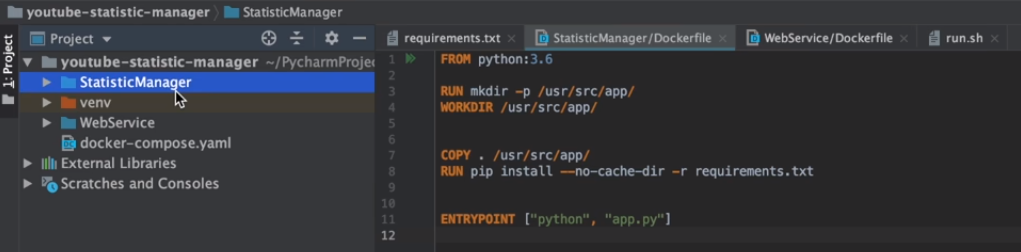
Команды:

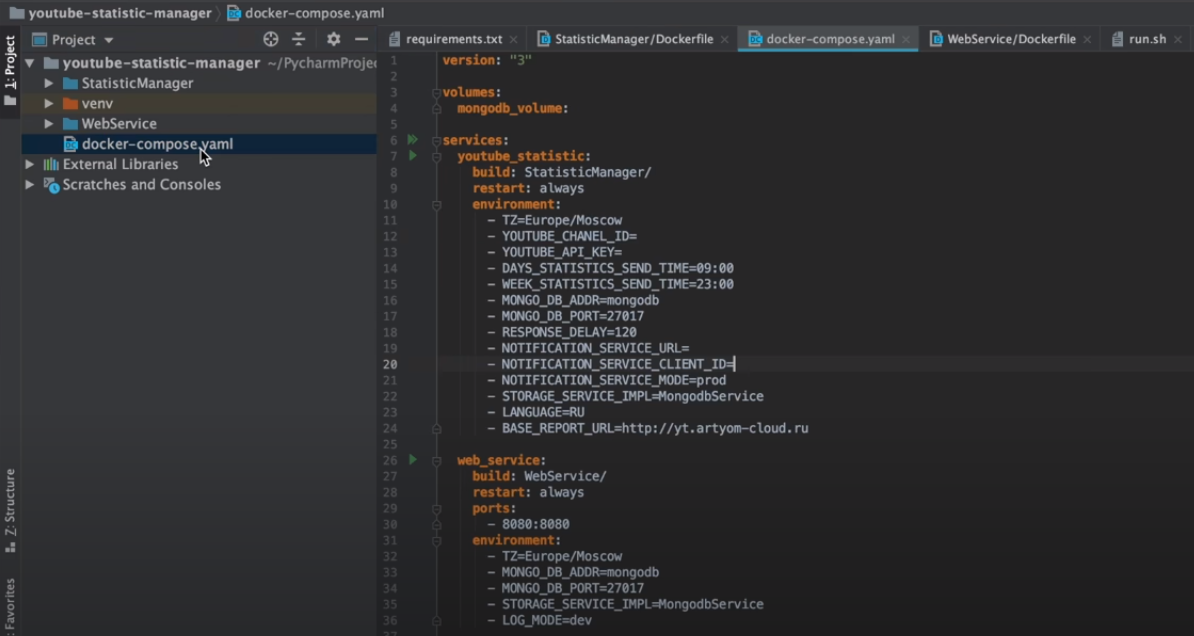
* docker-compose **up -d** – находясь в этот момент в папке с файлом “docker-compose.yml” мы запустим все описанные в файле “docker-compose.yml” контейнеры в фоне;
* docker-compose **stop -** останавливает все сервисы,
* docker-compose **down** – останавливает все сервисы, связанные с определённой конфигурацией Docker Compose. В отличие от команды **stop**, она также удаляет все контейнеры и внутренние сети, связанные с этими сервисами — но НЕ указанные внутри тома.
* останавливает и удаляет все работающие контейнеры и кастомную*(специально созданная под работу этих контейнеров)* сеть;
* docker-compose up -d **--build** – если мы, что-то изменили в наших файлах и хотим опять запустить наши микросервисы с этими изменениями;
* docker logs **имя\_работающего/остановленного\_Контейнера** – если мы хотим посмотреть логи какого-то конкретного контейнера;

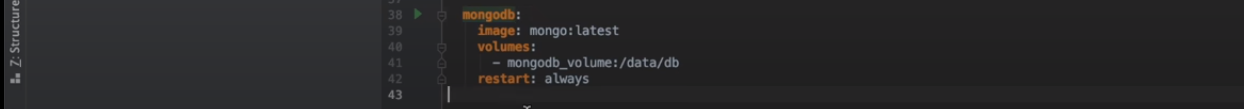






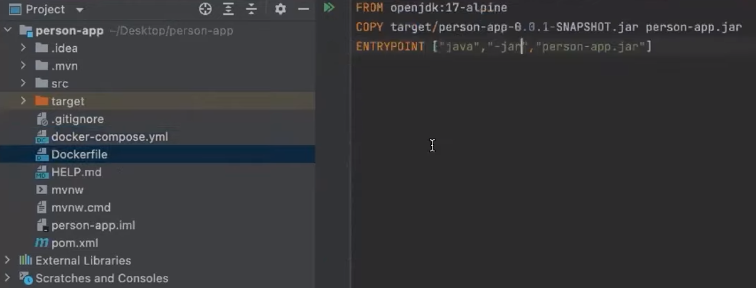




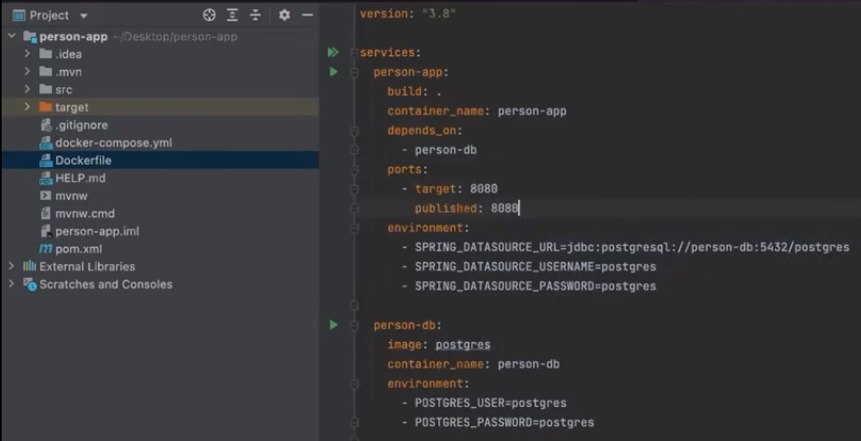


Видео <https://www.youtube.com/watch?v=O0pTOjWlKbc>:

Dockerfile: *(тут он вначале в Maven-не сделал команду “package”, создав в папке “Target” jar-файл проекта)*



docker-compose.yml:



, где:

- **version** - версия самого docker-compose;

- **build:** - для создания контейнеров из собственных образов;

- **image:** - для создания контейнеров из готовых образов с DockerHub;

**- depends\_on:** - тут указываем название всех сервисов от которых данный сервис зависит(т.е. требует запустить вначале те контейнеры);

**- port:** - это порт для связи по localhost;

**- expose:** - это порт для связи с внешними компами *(не loclalhost)*.

KAFKA\_BROKER\_ID — свойство broker.id — это уникальное и постоянное имя каждого узла в кластере.

KAFKA\_AUTO\_CREATE\_TOPICS\_ENABLE – если значение истинно, то оно позволяет брокерам создавать темы, когда на них впервые ссылается производитель или потребитель. Если для значения установлено значение false, тему следует сначала создать с помощью команды Kafka, а затем использовать.

KAFKA\_ZOOKEEPER\_CONNECT — указывает Кафке, как связаться с Zookeeper.

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR –показывает, на сколько брокеров-последователей будут скопированы данные с ведущего-лидера. Требуется, когда мы работаем с кластером с одним узлом. Если у вас три или более узлов, мы можем использовать значение по умолчанию.

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP –определяет пары ключ/значение для протокола безопасности, которые будут использоваться для каждого имени прослушивателя.

KAFKA\_ADVERTISED\_LISTENERS — это то, что будет возвращать брокер кафки клиентам, говоря как с ним (или, в зависимости от темы, с необходимым клиенту брокером) связаться и делает Kafka доступным снаружи контейнера, объявляя его местоположение на хосте Docker.