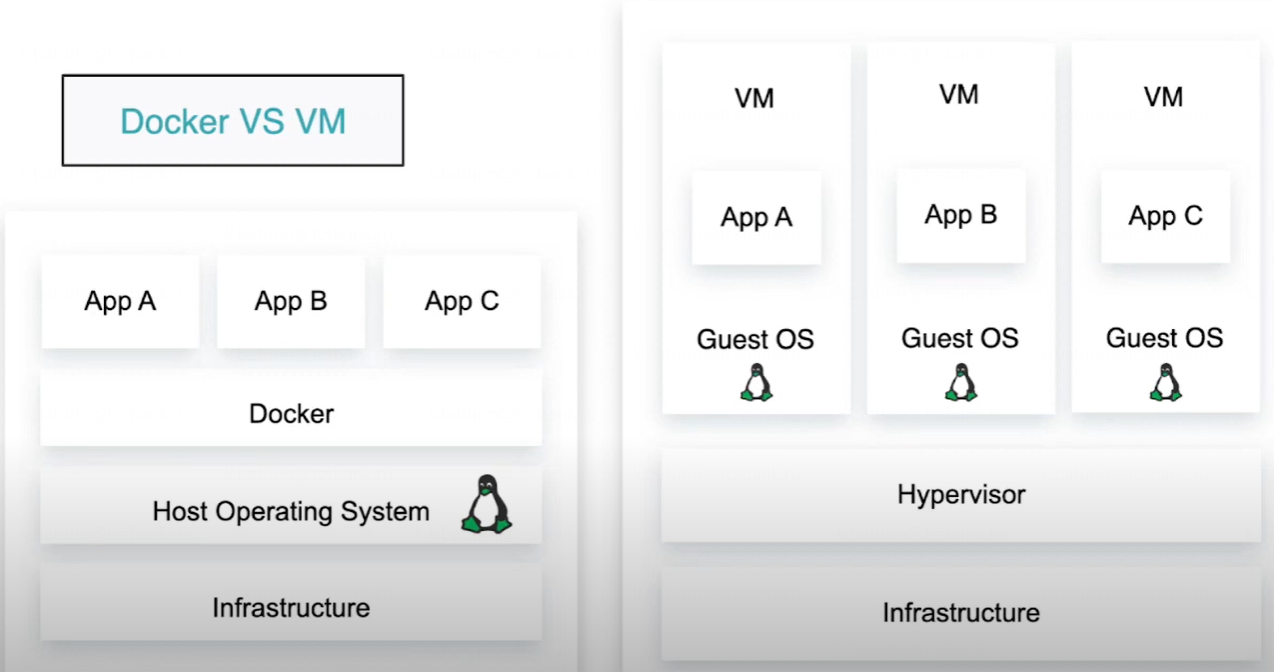
***Docker*** – это сервис для запуска приложений в контейнерах. Он обеспечивает стандартизацию контейнеров и исключает возможные конфликты с другим ПО или настройками ОС сервера. Для запуска на проде *docker* используется совместно с ***kubernetes***, который управляет запуском разных контейнеров *docker* на разных серверах.



**Преимущества *Docker* над *VM*:**

1. *VM* запускает отдельную копию ОС для запуска каждого приложения. *Docker* для работы требует единственную копию ОС (контейнеры разделяют одно ядро), поэтому его легко масштабировать путем увеличения количества контейнеров.
2. Контейнеры в *Docker* являются изолированной средой, а также запускаются как отдельный процесс в пользовательском пространстве.
3. *Docker* является портативным (не зависит от железа и ПО сервера), поэтому его удобно использовать в процессе разработки приложений.
4. *VM* загружаются медленнее (несколько минут), чем контейнеры *Docker* (несколько секунд).
5. *VM* требуют больше памяти (Гб), чем *Docker* (Мб). Как правило в *Docker* используют *alpine*-образы.

Нативной ОС для *Docker* является *Linux*, поэтому запуск на *Windows* должен происходить внутри виртуальной машины с ОС *Linux*. Десктопная версия работает на основе гипервизора *Hyper-V* или подсистемы *Windows Subsystem Linux*, а виртуальная – на основе подсистемы *Linux* (*WSL2*).

Список команд для установки *WSL2*:

*// включить "Подсистема Windows для Linux" через PowerShell*

*Enable-WindowsOptionalFeature -Online -NoRestart -FeatureName Microsoft-Windows-Subsystem-Linux*

*// включить "Платформа виртуальной машины" через PowerShell*

*Enable-WindowsOptionalFeature -Online -NoRestart -FeatureName VirtualMachinePlatform*

*// скачать версию WSL 1.2.5 с* [*https://github.com/microsoft/WSL/releases*](https://github.com/microsoft/WSL/releases) *с расширением msixbundle*

*// запустить PowerShell от имени администратора и установить пакет*

*Add-AppxPackage имя\_дистрибутива*

*// запустить PowerShell от имени пользователя и установить пакет*

*Add-AppxPackage имя\_дистрибутива*

*// перезагрузить ПК*

*// набрать в cmd команду "WSL” и убедиться в отсутствии установленных дистрибутивов*

*// задать WSL2 в качестве версии по умолчанию*

*wsl --set-default-version 2*

Список команд для установки *Linux* на *WSL*:

*// запросим список доступных дистрибутивов Linux*

*wsl -l -o*

*// установим нужный дистрибутив Linux*

*wsl --install -d Ubuntu-22.04*

*// проверим список установленных дистрибутивов Linux и их версию WSL с помощью команды в cmd*

*wsl -l -v*

Список команд для установки *Docker* и *Docker Compose* на *Linux*:

*// обновить список пакетов*

*sudo apt update*

*// установить необходимые пакеты для загрузки через HTTPS*

*sudo apt install apt-transport-https ca-certificates curl software-properties-common*

*// добавить ключ GPG для подтверждения подлинности в процессе установки (выведется ОК)*

*curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -*

*// скачивать только стабильные версии из репозитория Docker*

*sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"*

*// снова обновить список доступных пакетов*

*sudo apt update*

*// установить Docker Community Edition и Docker Compose*

*sudo apt install docker-ce docker-compose -y*

*// добавим текущего пользователя в группу "docker", чтобы не писать sudo перед командами docker*

*sudo usermod -aG docker ${USER}*

*// применим изменение группы "docker"*

*su - ${USER}*

*// проверим наличие пользователя в группе "docker"*

*id -nG*

*// проверить, что Docker работает*

*service docker status*

*docker run hello-world*

Список команд для настройки работы *TestContainers* через *WSL*:

*// внутри WSL создаем файл override.conf*

*sudo mkdir -p /etc/systemd/system/docker.service.d*

*sudo vim /etc/systemd/system/docker.service.d/override.conf*

*// добавляем в этот файл строки ниже*

*[Service]*

*ExecStart=*

*ExecStart=/usr/bin/dockerd --host=tcp://0.0.0.0:2375 --host=unix:///var/run/docker.sock*

*// нажимаем комбинацию* ***shift + q*** *и пишем* ***:exit***

*// перезапускаем WSL*

*wsl --shutdown*

*// внутри WSL ставим утилиту прослушки портов*

*sudo apt install net-tools*

*// проверяем, что порт 2375 в статусе LISTEN (иногда нужно запустить любой контейнер)*

*netstat -nl | grep 2375*

*// добавляем новое свойство в переменные среды Windows*

*// значение ${WSL\_ip} можно узнать либо командой в WSL "ifconfig eth0" либо в реестре Windows*

*// Компьютер\HKEY\_CURRENT\_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Lxss\NatIpAddress*

*DOCKER\_HOST=tcp://${WSL\_ip}:2375 // заменить на ip-адрес WSL*

*// опционально, выключить внутри WSL ipv6*

*sudo sysctl -w net.ipv6.conf.default.disable\_ipv6=1*

*sudo sysctl -w net.ipv6.conf.all.disable\_ipv6=1*

*// опционально, в профиле пользователя Windows создать файл* ***.testcontainers.properties*** *с наполнением*

*docker.client.strategy = org.testcontainers.dockerclient.EnvironmentAndSystemPropertyClientProviderStrategy*

*docker.host = tcp://[::1]:2375*

*docker.cert.path = \\wsl$\home\SeuUserLinux\.docker*

*ryuk.container.privileged = false*

***Dockerfile*** – специальный файл без расширения в корне приложения, который содержит инструкции для создания *docker-image*. Из одного *dockerfile* создается один *docker-image*.

***Image*** – это шаблон (набор слоев), который содержит все необходимые файлы для создания и автономной работы контейнера.

Сборка образа происходит послойно и кэшируется после каждой изменения. Изменить существующие слои нельзя, можно только создать новые слои поверх старых. Такая архитектура позволяет переиспользовать слои в разных образах и ускорить процесс их сборки.

***Container*** – это рабочее окружение, которое создается из образа. Рекомендовано запускать один контейнер под одну задачу. При завершении такой задачи, *docker* автоматически остановит её контейнер.

Все данные, накопленные за время работы контейнера, уничтожаются после его удаления или перезагрузки. Эту проблему решает ***volume*** – внешняя директория, доступная одному или нескольким контейнерам.

***Repository*** – это хранилище различных версий образов. Для именования версий образов используются один или несколько **тегов**. Для именования самой маленькой версии образа используется тег ***alpine***.

***Registry*** – это хранилище репозиториев. Реестры могут быть локальными и удаленными. Самым популярным удаленным репозиторием является ***DockerHub***.

Рассмотрим инструкции, используемые для написания ***Dockerfile***. Для этого используется **императивный подход** – последовательно прописываются все действия. Каждая инструкция пишется заглавными буквами, а их выполнение происходит последовательно.

* Установить аргументы, которые будут заданы при сборке образа. Инструкция всегда должна быть первой.

*ARG <ArgimentName>=<DefaultValue>*

* Выбрать базовый образ, на основе которого создается текущий образ.

*FROM <image>:<tag>*

Необязательный параметр ***tag*** указывает конкретную версию образа. По умолчанию ***latest***.

* Задать метаданные для образа.

*LABEL version 1.0*

* Создаем и задаем рабочую директорию в образе. С этой директорией будут работать последующие команды.

*WORKDIR <директория\_внутри\_образа>*

* Скопировать файлы и директории из указанной локальной директории в директорию внутри образа (создастся новый слой). Если целевой директории не существует, то она создается.

*COPY <локальная\_директория> <путь\_внутри\_образа>*

* Скопировать файлы и директории из указанной локальной директории в директорию внутри образа и автоматически распаковать архивы (создастся новый слой). Если целевой директории не существует, то она создается. Не рекомендуется использовать из-за возможной *zip*-бомбы.

*ADD <локальная\_директория> <путь\_внутри\_образа>*

* Задать переменные среды окружения внутри образа

*ENV TZ=”Europe/Moscow” // установить внутри образа часовой пояс +3*

* Задать набор команд, которые выполняются при сборке образа (создастся новый слой). Команды можно переносить на следующую строку с помощью символа **\** (обратный слеш). Каждую строку с командой можно выполнить отдельно с помощью символа **|** (вертикальная черта).

*RUN npm install npm@8.19.4 --silent // установить пакет npm версии 8.19.4*

Не рекомендуется задавать секреты в инструкциях ARG, *COPY*, *ADD* и *RUN*, т.к. они отобразятся в слое.

* Задать порты, которые могут быть открыты

*EXPOSE <номер\_порта>*

* Задать набор команд с аргументами, которые выполняются при запуске контейнера. Аргументы нельзя переопределить через параметры командной строки. *PID*-процесс запущенный таким образом не будет отображаться в диспетчере.

*ENTRYPOINT [<command>, <param1>, <param2>]*

* Задать набор команд с аргументами, которые выполняются при запуске контейнера. Аргументы могут быть переопределены.

*CMD [<command>, <param1>, <param2>]*

* Указать место для постоянного хранилища файлов.

*VOLUME <VolumeName>*

Рассмотрим самые популярные команды для работы с ***Docker***:

* Вывести список всех команд

*docker*

* Вывести информацию о версии докера

*docker --version*

* Вывести список всех образов

*docker image ls*

*docker images // альтернативная команда*

* Создать образ из *Dockerfile*

*docker build . -t <ImageName>:<Tag>*

***Точка*** внутри команды – это путь до *Dockerfile*, на основе которого производится сборка.

Необязательный ключ ***-t*** позволяет задать имя и тег образу. При использовании тега ***latest***, предыдущий образ с таким же тегом будет перезаписан.

Необязательный ключ ***--build-arg*** позволяет передать значение для ***ARG*** внутри *Dockerfile*.

* Выполнить поиск образа на *DockerHub* по части его имени или тегу

*docker search <ImageName>*

* Скачать образ с *DockerHub* по его имени или тегу

*docker pull <ImageName>*

* Загрузить образ на *DockerHub* в существующий репозиторий пользователя. Имя образа может содержать только строчные буквы, цифры, дефисы (не с начала) и нижние подчеркивания. **Тэг** – это необязательная часть имени, например, версия. По умолчанию все загружаемые образы публичные.

*docker login -u <UserName> -p <Password> // авторизация на DockerHub*

*docker push <UserName>/<ImageName>::<Tag>*

* Вывести информацию об образе по его имени или *image\_id*

*docker image inspect <ImageName>*

*docker inspect <ImageName> // альтернативная команда*

* Удалить образ по его имени или *image\_id*

*docker image rm <ImageName>*

* Удалить все неиспользуемые образы

*docker image prune*

* Вывести список всех томов

*docker volume ls*

* Вывести информацию о томе по его имени

*docker volume inspect <VolumeName>*

*docker inspect <VolumeName> // альтернативная команда*

* Создать новый том

*docker volume create <VolumeName>*

Необязательный ключ ***--tempfs*** позволяет записывает данные в памяти без сохранения информации при удалении контейнера.

* Удалить том по его имени

*docker volume rm <VolumeName>*

* Удалить все неиспользуемые тома

*docker volume prune*

* Вывести список всех контейнеров

*docker container ls -a*

*docker ps -a // альтернативная команда*

* Вывести список всех запущенных контейнеров

*docker container ls*

*docker ps // альтернативная команда*

* Создать и запустить новый контейнер из образа. Если образ отсутствует локально, будут выполнены поиск и загрузка c *DockerHub*.

*docker container run <ImageName>*

*docker run <ImageName> // альтернативная команда*

*docker run <ImageName>:<Tag> \ // полная команда с разбивкой на строки*

*--name <ContainerName> \*

*-p 8080:8080 \*

*-d \*

*--rm \*

*-v <VolumeName>:<Container\_path>*

Необязательный ключ ***--name*** позволяет задать имя контейнеру. В случае его отсутствия, *Docker* сгенерирует имя самостоятельно.

Необязательный ключ ***-v <host\_path>:<container\_path>*** позволяет настроить ***mapping*** (проброс) томов. Этот ключ также можно использовать для подключения существующего тома по его имени ***-v <VolumeName>:<container\_path>***. Если том не будет найден, то он создастся.

Необязательный ключ ***--mount*** совместно с *volume* позволяет копировать в том имеющийся путь *container\_path*.

Необязательный ключ ***-p <host\_port>:<container\_port>*** позволяет настроить ***mapping*** (проброс) портов.

Необязательный ключ ***-d*** позволяет не занимать текущий процесс консоли запущенным приложением.

Необязательный ключ ***--rm*** позволяет удалить контейнер при его остановке.

* Запустить существующий контейнер по его имени или *container\_id*

*docker container start <ContainerName>*

*docker start <ContainerName> // альтернативная команда*

* Остановить контейнер по его имени или *container\_id*

*docker container stop <ContainerName>*

*docker stop <ContainerName> // альтернативная команда*

* Принудительно остановить зависший контейнер по его имени или *container\_id*

*docker container kill <ContainerName>*

*docker kill <ContainerName> // альтернативная команда*

* Вывести информацию о контейнере по его имени или *container\_id*

*docker container inspect <ContainerName>*

*docker inspect <ContainerName> // альтернативная команда*

* Удалить контейнер по его имени или *container\_id*

*docker container rm <ContainerName>*

*docker rm <ContainerName> // альтернативная команда*

* Удалить все остановленные контейнеры

*docker container prune*

* Вывести логи контейнера по его имени или *container\_id*

*docker logs <ContainerName> -f*

Необязательный ключ ***-f*** включает режим непрерывного просмотра (все изменения будут отображаться автоматически).

* Подключиться к работающему контейнеру по его имени или *container\_id* для последующего выполнения команд в нем.

*docker exec <ContainerName>* */bin/sh -it*

*docker exec <ContainerName> /bin/bash -it // альтернативная команда для другой ОС*

Необязательный ключ ***-i*** включает интерактивный режим. Необязательный ключ ***-t*** включает режим терминала. Для выхода из этого режима можно нажать ***Ctrl + C*** (контейнер остановится) или набрать в терминале ***exit*** (контейнер продолжит работать).

* Создать новый образ на основе текущего контейнера по его имени или *container\_id*

*docker container commit <ContainerName> <NewImageName>:<NewTag>*

*docker commit <ContainerName> <NewImageName>:<NewTag> // альтернативная команда*

Подробнее про команды *docker* можно почитать [тут](https://docs.docker.com/reference/).

Рассмотрим пример входа в ***postgres*** внутри запущенного контейнера:

*docker exec PostgresContainer bash -it // подключаемся к контейнеру PostgresContainer*

*psql -U UserName // запускаем postgres от имени UserName*

*\l // выводим список созданных БД*

*\c DbName // подключаемся к БД DbName*

*\d // выводим информацию о текущей БД*

*\d TableName // выводим информацию о таблице TableName*

Рассмотрим процесс создания и запуска *docker*-контейнера для ***java*** приложения:

1. Cобираем текущее приложение в *jar*-архив в папке *target* с помощью команды *“mvn clean package”*.
2. Запускаем приложение из *jar*-архива и проверяем его работоспособность с помощью команды *“java -jar target/[имя\_архива].jar”*. останавливаем его командой *Ctrl+C*.
3. Добавляем в корень проекта текстовый файл без расширения с именем *Dockerfile*.
4. Добавляем в *Dockerfile* инструкции для сборки.

*FROM amazoncorretto:11-alpine-jdk // ОС Linux с java 11 от Amazon Correto*

*COPY target/\*.jar app.jar // скопировать все jar-файлы в контейнер*

*ENTRYPOINT ["java","-jar","/app.jar"] // запустить java-приложение после старта контейнера*

1. Собираем собственный образ *dock\_image*.

*docker build -t dock\_image .*

1. Создаем и запускаем контейнер *dock\_container* из образа *dock\_image*.

*docker run --name dock\_container -p 8080:8080 dock\_image*

Рассмотрим процесс создания и запуска *docker*-контейнера для ***react*** приложения:

1. Добавляем в корень проекта текстовый файл без расширения с именем .d*ockerignore*. Добавляем в него файлы, которые не будут использоваться *docker* при создании образа:

*node\_modules  
Dockerfile  
.env  
.dockerignore  
.DS\_Store*

1. Добавляем в корень проекта текстовый файл без расширения с именем *.env*. Добавляем в него настройки для переменных сред:

*VITE\_API\_BASE\_URL=http://localhost:8080*

1. Добавляем ***--host*** в файл *package.json* для работы на локальном хосте

*“scripts”:“dev”: “vite --host”*

1. Добавляем в корень проекта текстовый файл без расширения с именем *Dockerfile*.
2. Добавляем в *Dockerfile* инструкции для сборки.

*FROM node:19-alpine // ОС Linux с установленной node*

*ARG api\_base\_url // принимаем извне это значение  
WORKDIR /app // создаем целевую рабочую директорию  
COPY package\*.json . // копируем json файлы с зависимостями в рабочую директорию  
RUN npm i --silent // скачиваем и устанавливаем все библиотеки-зависимости  
COPY . . // копируем весь код проекта, кроме запрещенного .dockerignore  
RUN echo "VITE\_API\_BASE\_URL=${ api\_base\_url }” > .env // переопределяем переменные среды  
EXPOSE 5173 // открываем порт 5173  
CMD ["npm", "run", "dev"] // запускаем фронтенд при старте контейнера*

1. Собираем собственный образ *dock\_image*.

*docker build . -t straigt/straigt-react --build-arg=http:localhost:8088*

1. Создаем и запускаем контейнер *dock\_container* из образа *dock\_image*.

*docker run --name dock\_container -p 3000:5173 dock\_image*

Секретные ключи, доступы и токены не следует хранить в коде. Передать их в контейнер можно несколькими способами:

* Прописать переменные окружения прямо в *Dockerfile* с помощью команды *ENV*.

*ENV token 12345*

* Задать переменные окружения *ENV* при сборке контейнера, выполнив команду run с ключом ***-e***

*docker run <имя образа> -e token=12345*

* Сохранить переменные окружения в настроечном файле *Docker Compose*.

Для загрузки значений из переменных окружения в *java* нужно изменить файл *application.properties*:

*jdbc.url=jdbc:postgresql://${DB\_HOST}:${DB\_PORT}/${DB\_NAME}*

*jdbc.username=${POSTGRES\_USER}*

*jdbc.password=${POSTGRES\_PASSWORD}*

Для отладки *java*-приложения, необходимо запустить его со следующими параметрами:

*-agentlib:jdwp=transport=dt\_socket,server=y,suspend=n,address=\*:[номер порта для отладки]*

*transport=dt\_socket* – в качестве способа подключения к *JVM* будут использоваться сокеты.

*server=y* – к приложению будут подключаться для отладки, а не наоборот.

*suspend=n* – *JVM* будет ждать, пока отладчик подключится для выполнения команд.

*address=\*:[номер порта]* – номер порта для подключения (обычно +1 от используемого порта)

В результате финальная версия *Dockerfile* будет выглядеть так:

*FROM amazoncorretto:11-alpine-jdk*

***ENV JAVA\_TOOL\_OPTIONS -agentlib:jdwp=transport=dt\_socket,server=y,suspend=n,address=\*:8081***

*COPY target/\*.jar app.jar*

*ENTRYPOINT ["java","-jar","/app.jar"]*

Запустим контейнер, пробросив уже два порта:

*docker run --name dock\_container -p 8081:8081 -p 8080:8080 dock\_image*

Теперь можно запустить в *IDEA* удаленный дебаг приложения.



Для запуска и управления взаимодействием нескольких контейнеров используется ***Docker Compose***. Например, в одном контейнере работает сервис, а в другом – база данных. Для этого используется **декларативный подход** – прописывается финальный результат.

Для запуска проекта используется команда ***docker-compose up***. Необязательный ключ ***--build*** заставляет *docker-compose* заново создать все образы, используемые в проекте.

Для остановки проекта используется команда ***docker-compose stop***.

Для остановки проекта и удаления текущих контейнеров используется команда ***docker-compose down***.

Команда ***docker-compose pull*** загружает актуальную версию проекта из *DockerHub*.

Инструкции по развертыванию проекта в нескольких контейнерах пишут в файле ***docker-compose.yml***. Рассмотрим пример такого файла:

*# версия Docker Compose*

*version: '3.8'*

*# придумываем любые имена для контейнеров, которые должны быть развёрнуты*

*services:*

*# описание контейнера db*

*db:*

*# загружаем образ из DockerHub и запускаем контейнер*

*image: postgres:13.7-alpine*

*# переменные окружения*

*environment:*

*- POSTGRES\_DB=later*

*- POSTGRES\_USER=root*

*- POSTGRES\_PASSWORD=root*

*# используем сеть*

*networks:*

* + *webnet*

*# пробрасываем том db-data в директорию внутри контейнера*

*volumes:*

*- db-data:/var/lib/postgresql/data/*

*# описание контейнера api*

*api:*

*# создаем образ на основе локального dockerfile и запускаем контейнер*

*build: .*

*# пробрасываем порты*

*ports:*

*- "8088:8080"*

*# определяем после какого контейнера должен запускаться текущий*

*depends\_on:*

*- db*

*# перезапускаем, если контейнер упадет*

*restart: on-failure*

*# переменные окружения*

*environment:*

*- DB\_NAME=later*

*- POSTGRES\_USER=root*

*- POSTGRES\_PASSWORD=root*

*- DB\_HOST=db*

*- DB\_PORT=5432*

*# используем сеть*

*networks:*

* + *webnet*

*# пробрасываем текущую тайм-зону в контейнер*

*volumes:*

* + */etc/timezone:/etc/timezone:ro*
  + */etc/localtime:/etc/ localtime:ro*

*# описание контейнера react*

*react:*

*# ЛИБО используем уже существующий контейнер*

*container\_name: react*

*# ЛИБО создаем образ на основе локального dockerfile и запускаем контейнер*

*build:*

*# путь до dockerfile*

*context: frontend/react*

*# передаем аргументы для переменных сред внутрь dockerfile*

*args:*

*api\_base\_url:* [*http://localhost:8088*](http://localhost:8088)

*# пробрасываем порты*

*ports:*

*- "3000:5173"*

*# определяем после какого контейнера должен запускаться текущий*

*depends\_on:*

*- api*

*# перезапускаем, если упадет контейнер*

*restart: unless-stopped*

*# создаем свой именованный том db-data*

*volumes:  
 db-data:*

*# создаем свою сеть webnet*

*networks:*

*webnet:*

*driver: bridge*

Подробнее про модульные приложения можно почитать [тут](https://spring.io/guides/gs/multi-module/).

Рассмотрим **приемы работы в командной строке**:

*<консольная команда> --help // вывести справку по команде*

*hostname // вывести имя хоста*

*hostname -i // вывести ip-адрес хоста*

*pwd // вывести путь до текущей директории*

*ls // вывести содержимое текущей директории*

*ls - l // вывести содержимое текущей директории, включая скрытые*

*ls -a // вывести содержимое текущей директории и информацию о них*

*ls -t // вывести содержимое текущей директории, отсортировав по возрастанию даты*

*ls <путь до директории> //вывести содержимое другой директории*

*cd licenses // переход в другую директорию*

*cd / // возврат в корневую директорию*

*cd ./dev // переход в директорию dev в текущем каталоге*

*cd .. // возврат на один уровень выше*

*cd ../folder2 // возврат на один уровень выше и провалиться в соседнюю директорию*

*touch answer.txt // создать один файл. Если файл существует – обновить дату редактирования*

*touch answer.txt pom.xml // создать два файла*

*touch new\_folder/answer.txt // создать один файл в другой директории*

*mkdir new\_folder // создать папку внутри текущей директории*

*mkdir /var/log/temp // создать папку внутри по указанному пути*

*rm file1.txt // удалить файл*

*rm file1.txt file2.txt // удалить два файла*

*rm folder/file1.txt // удалить файл из директории*

*rm -r folder // удалить папку рекурсивно со всем содержимым*

*cp logs.txt double\_logs.txt // переименовать файл logs.txt в double\_logs.txt*

*cp ../logs/logs.txt double\_logs.txt // скопировать файл в текущую директорию и назвать по-новому*

*cp -r ../docs/ ../Documents/ // скопировать папку со всем содержимым*

*mv card.txt / // перенести файл из текущей директории в корневую*

*mv card.txt /home/logs/2020 // перенести файл из текущей директории в указанную*

*mv my\_app.ssh you\_app.sh // переименовать файл*

*echo “Ау!” // Вывод текста в терминал*

*echo “Testing” > secret.txt // записать в него строку в файл. Если файла нет – он создастся*

*cat 1.txt 2.txt 3.txt // вывод содержимого трех файлов*

*cat a.txt > b.txt // стереть файл b.txt и записать в него содержимое файла a.txt*

*cat a.txt >> b.txt // записать содержимое файла a.txt в конец файла b.txt*

*cat 1.txt 3.txt >> b.txt // записать содержимое обоих файлов 1.txt и 2.txt в конец файла b.txt*

Для редактирования текста внутри контейнера можно использовать ***nano*** редактор. Клавиша *Ctrl* в нем обозначена символом *^*.

Специальные сочетания клавиш можно увидеть, нажав *Ctrl+G* в запущенном редакторе.

Сохранить файл можно сочетанием клавиш *Ctrl+O*.

Выйти из редактора можно сочетанием клавиш *Ctrl+X*.

*apk add nano // установить редактор nano*

*nano <название или путь к файлу> // открыть файл в редакторе*

*grep <ключи> <искомое слово/фраза> <путь до файла> // поиск текста внутри файла*

*grep -i ERROR logs.txt > errors.txt // найти строки с “ERROR”, игнорируя регистр, и сохранить в файл*

Команда ***grep*** чувствительна к регистру искомого текста.

Ключ ***-i*** позволяет игнорировать регистр.

Ключ ***-n*** дополнительно отобразит номер строки.

Ключ ***-c*** вернет количество строк, которое соответствует условиям поиска.

Ключ ***-B*** (*--before-context*) задает количество отображаемых строк до искомой.

Ключ ***-A*** (*--after-context*) задает количество отображаемых строк после искомой.

Ключ ***-C*** (*--context*) задает количество отображаемых строк до и после искомой одновременно.

*tail <имя файла> // вывести последние 10 строк файла*

*tail -5 <имя файла> // вывести последние 5 строк файла*

*tail -f <имя файла> // вывести последние 10 строк файла в режиме непрерывного просмотра*

Ключ ***-f*** включает режим непрерывного просмотра. Все изменения будут отображаться автоматически.

*ps // вывод списка запущенных процессов*

*ps | grep nano // вывод списка запущенных процессов редактора nano*

*top // вывод списка запущенных процессов в реальном времени*

*top | grep nano // вывод списка запущенных процессов редактора nano в реальном времени*

*fuser -k 9999/tcp // остановить процесс, использующий порт 9999 по протоколу tcp*

*kill [ключ] [PID] // остановить процесс с указанным номером PID*

Ключ ***-9*** делает команду принудительной. Подробнее [тут](https://pubs.opengroup.org/onlinepubs/9699919799/utilities/kill.html#tag_20_64).

*chmod +x script.sh // добавить права для выполнения файла*

*ssh kbaturin:password:51.250.15.136 // подключиться через ssh*

*// проброс портов через ssh (при обращении к порту 8081 перенаправлять их на порт 8080)*

*ssh kbaturin:password:51.250.15.136 -N -L 8081:51.250.15.136:8080*

*// проброс портов через ssh и через промежуточный сервер*

*ssh kbaturin:password:51.250.15.136 -N -L 8081:51.250.15.136:8080 51.250.84.187*