# Контейнеры

**Контейнеризация** - метод виртуализации, при котором ядро операционной системы поддерживает несколько изолированных экземпляров пространства пользователя вместо одного. Эти экземпляры (обычно называемые контейнерами или зонами) с точки зрения выполняемых в них процессов идентичны отдельному экземпляру операционной системы.

**Пространство пользователя -** адресное пространство виртуальной памяти операционной системы, отводимое для пользовательских программ.

Контейнеры представляют собой средства инкапсуляции приложения с его зависимостями. контейнер содержит изолированный экземпляр операционной системы (ОС), который можно использовать для запуска приложений.

Контейнеры это изолированные процессы, выполняющиеся на одном хосте.

Ядро обеспечивает полную изолированность контейнеров, поэтому программы из разных контейнеров не могут воздействовать друг на друга.

Изоляция осуществляется благодаря двум механизмам ядра Linux – *пространствам имен (namespaces)* и *контрольным группам (cgroups).*

**Пространство имён(namespaces) – функция ядра linux**, очень полезна при изоляции групп процессов друг от друга. Идея namespaces заключается в том, чтобы обернуть определенные глобальные системные ресурсы в слой абстракции. Пространства имен определяют набор ресурсов, которые может использовать процесс. Т.о. создается впечатление, что процессы внутри пространства имен имеют свой собственный изолированный экземпляр ресурса. Пространства имен ядра позволяют группам процессов иметь различные представления о системе.

В Linux имеется шесть стандартных пространств имён:

* Mnt (файловая система, mount) – изолирует файловую систему, от других процессов и от хоста.
* IPC – изоляция для механизмов взаимодействия процессов, таких как семафоры, очереди сообщений, разделяемая память и т.д.
* net – изоляция сетевого стека хоста. Ограничивает видимость процесса внутри сети.
* USER namespace(usr) – изоляция ID пользователей системы. Например процесс может иметь root привилегии внутри пространства имен, а вне его нет.
* pid – изоляция дерева системных процессов. В линукс есть один корневой процесс, который запускает остальные. В namespace другие процессы могут воспринимать себя, как корневые. Процессы могут взаимодействовать друг с другом только в рамках одного namespace.
* **UNIX Time-sharing System (uts)** – изолирует имя хоста системы для определенного процесса.

При помощи **cgroups**, ОС Linux может с лёгкостью управлять и отслеживать выделение ресурсов для выбранного процесса и устанавливать ограничения на использование таких ресурсов, как ЦП, Оперативная память, пропускная способность в сети.

Каждый процесс может воспринимать только одно пространство имен. Например, когда определенный процесс хочет посмотреть структуру каталогов, то ядро показывает ему копию файловой системы для пространства имен, к которому он принадлежит в данный момент. Если процесс запросит свой ID в системе, то ядро сообщит ID в его текущем пространстве имен (в случае со вложенными пространствами).

У каждого контейнера будет своё пространство имён и процессы, запущенные внутри этого пространства имён, и эти контейнеры не будут иметь доступа к чему-либо, что находится снаружи их пространства имён.

Преимущества:

* Устранение ошибок, обусловленных изменениями рабочей среды. Теперь нельзя будет сказать “это работает на моем компьютере”.
* Легче чем виртуалка. Каждая VM имеет свои собственные ресурсы, такие как ядро ОС, файловая система и сетевые интерфейсы. При этом виртуалке сразу выделяется какое-то количество оперативной памяти, и она забирает его все себе (Даже если фактически использует меньше). Контейнеры используют общую операционную систему хоста и разделяют ядро ОС. Они так же разделяют и оперативную память, и берут не больше, чем нужно для выполнения. Вообще это как обычные процессы, только в изолированных адресных пространствах.  
  Можно запускать одновременно десятки контейнеров, и имитировать возможность распределенной системы.
* Изоляция. Каждый контейнер работает в своей собственной виртуальной среде, что позволяет избежать конфликтов между зависимостями. Приложения в контейнере не влияют друг на друга. Нет ситуации когда одно приложение захватило все ресурсы, а другое простаивает.
* Масштабируемость. Легко масштабировать приложение горизонтально, запуская несколько контейнеров.

Docker изначально создавался под Linux. Поэтому на Windows и macOS запускают виртуальную машину с Linux, а поверх неё — Docker. В macOS используют VirtualBox, а в Windows — Hyper-V.

Если цель виртуалки — полностью воспроизвести устройство компьютера, то основная цель Docker — создать среду для одного приложения

# **Состояния контейнера**:

* создан (created) – контейнер инициализирован командой *docker create*, но его работа пока еще не началась.
* перезапуск (restarting)
* активен или работает (running)
* приостановлен (paused)
* остановлен (exited)

# docker run

**–** команда запускающая контейнер.



Опции:

* **-h, --hostname** – задать хост
* **--name** – задать имя контейнера
* **--rm** – автоматически удалить контейнер после его остановки. Так же будут удалены все связанные с ним ресурсы, такие как файлы системы контейнера.
* **-d**, **--detach** – запустить контейнер в фоновом режиме. (Без этого флага мы попадем в терминал контейнера, и после закрытия этого терминала контейнер остановится.)
* **-v, --volume**  – запуск контейнера с volume.
* --**volumes-from –** монтирует в контейнер тома из указанного контейнера.
* -**p, --publish** – пробросить порты из контейнера на хостовую машину. Первый это порт хоста, второй виртуалки.  
  
* **-P, --publish-all** – Объявляет все порты, открываемые в контейнере, доступными на хосте. Механизм Docker должен автоматически выбрать свободные порты для перенаправления с хоста в контейнер. Чтобы определить номера портов, назначенные Докером, можно выполнить команду **docker port <container>**
* **-a, --attach –** позволяет присоединиться к выводу контейнера и просматривать его логи в реальном времени.
* -**i, --interactive –** запустить контейнер в интерактивном режиме. В этом режиме вы можете взаимодействовать с командной оболочкой контейнера, вводить команды и получать их вывод. При выходе из терминала (Ctrl+C) контейнер будет остановлен.
* **--restart –** используется для настройки поведения контейнера в случае его аварийной остановки или выхода из строя. Принимает несколько значений:

1. **no** (по умолчанию) – контейнер не будет автоматически перезапущен после остановки или сбоя.
2. **on-failure[:max-retries]** – контейнер будет перезапущен только в случае сбоя. Количество попыток перезапуска можно указать с помощью необязательного аргумента.
3. **always –** контейнер будет автоматически перезапущен, независимо от причины его остановки.
4. **unless-sopped –** контейнер будет автоматически перезапущен, пока его не остановят явно.

Если вы хотите изменить поведение перезапуска уже запущенного контейнера, нужно использовать команду *docker update*.

* **-t, --tty** – используется для выделения псевдо-терминала при запуске контейнера. Это позволяет вам взаимодействовать с командной оболочкой и получать вывод (только вывод) в режиме реального времени. Часто используется с ключом –*interactive*.
* -**e, --env –** определяет переменные среды внутри контейнера
* **--env-file** – используется для передачи переменных среды через заданный файл.
* --**expose** – определяет номер порта или диапазон номеров портов, предназначенных для использования в контейнере, но в действительности не открывает каких-либо портов. Применение этого ключа имеет смысл только в сочетании с ключом *-P*, а также при установке соединения между контейнерами. Имеет аналог в Dockerfile.
* **--link** – используется для связи двух контейнеров и создания сетевого соединения между ними. устарел и не рекомендуется к использованию в новых проектах.
* --**entrypoint –** перезаписать значение *ENTRYPOINT* в Dockerfile.
* -**u, --user –** пользователь, от имени которого выполняются команды. Переопределяет значение *USER* из Dockerfile.
* -**w, --workdir –** устанавливает рабочий каталог в контейнере. Переопределяет значение *WORKDIR* из Dockerfile.
* --**network –** указать в какой сети будет находиться контейнер.

# docker ps

**–** посмотреть список запущенных контейнеров. В списке отображается ID контейнера; команда, выполняемая внутри контейнера; пробрасываемые порты; docker-образ; время, прошедшее с момента запуска контейнера; статус контейнера; имя контейнера (удобно для чтения).

-**a, --all –** вывести все контейнеры

-**q, --quiet –** вывести только идентификаторы контейнеров. Может быть удобно для передачи с помощью конвейера в другие команды, например docker rm.

**--filter –** применить фильтры к выводу контейнеров.

-**s, --size –** показать размеры занимаемого дискового пространства контейнерами.

# docker logs <container> -

вывести логи контейнера.

-**f, --follow –** следить за логами в режиме реального времени.

-**t, --timestamps –** время записи

--**since –** выводить логи начиная с указанной даты и времени. Можно указывать абсолютно (*2013-01-02T13:23:37Z*) либо относительно (*42m* – за последние 42 минуты).

--**tail, -n –** показать последние n строк лога.

--**until –** используется с флагом -f чтобы выводить логи до наступления определенного момента времени.

# docker stop <container>

**-** остановить контейнер.При этом контейнер безопасно завершит свои текущие задачи, и сохранит все данные перед остановкой.

# docker rm <container>

**-** удалить контейнер.

**docker rm -v –** удалить контейнер и связанные с ним volumes.

-**f –** позволяет удалить работающие контейнеры.

# docker attach [OPTIONS] CONTAINER

– позволяет присоединиться к работающему контейнеру для взаимодействия с его командной оболочкой.

Использование комбинации ctrl + c для выхода завершит наблюдаемый процесс и приведет к завершению работы контейнера.

Если контейнер был запущен в фоновом режиме (с ключом *-d*), то attach присоединиться только к стандартному потоку вывода контейнера, и вы не сможете вводить команды в командную оболочку контейнера. В этом случае можно использовать *docker exec*.

# docker exec

**–** запускает заданную команду внутри контейнера. Позволяет вам запускать интерактивную оболочку, например bash, выполнять сценарии, устанавливать пакеты. Команды можно выполнять как в фоновом, так и в интерактивном режиме.

**-i, --interactive –** запуск интерактивного режима, который позволяет взаимодействовать с командной строкой контейнера.

**-t, --tty –** ассоциирует терминал хоста с командной строкой контейнера.

-**d, --detach –** запустить команду в фоновом режиме. Результат не будет выведен в терминал.

--**priveleged –** позволяет выполнить в контейнере операции, требующие повышенных привелегий.

-**u, --user –** имя пользователя или UID от имени которого будет выполняться команда.

**--workdir –** задает рабочий каталог, в котором будет выполняться команда внутри контейнера.

-**e, --env –** устанавливает переменные среды для команды.

Запуск интерактивной оболочки в контейнере:



Выполнение команды внутри контейнера:



# docker cp

**-** используется для копирования файлов и директорий между локальной машиной и контейнером Docker.



# docker kill

**–** посылает сигнал SIGKILL основному процессу контейнера, по которому выполняется немедленное завершение работы контейнера.

**-s –** послать контейнеру другой сигнал.

# docker pause

**–** приостановить выполнение всех процессов внутри контейнера. Может быть полезна при отладке.

# docker unpause

**–** продолжить выполнение процессов, после их остановки с помощью pause.

# docker restart

**–** перезапускает один или несколько контейнеров.

# docker info

**–** выводит информацию о системе Docker и хосте, на котором она работает, такую как: версия докера; название и версия операционной системы хоста; драйвер хранения, используемый Docker для управления данными контейнеров и образов; драйвер журналирования, используемый для записи логов контейнеров; количество запущенных, приостановленных и остановленных контейнеров; количество образов, сохраненных в вашей среде; количество доступных процессоров на вашей системе; объем доступной памяти на вашей системе; корневой каталог Docker.

# docker help

**–** справка по заданной команде. Аналогична флагу **–help**.

# docker version

– версия докера.

# docker diff

**–** показывает изменения в файловой системе контейнера по сравнению с файловой системой образа, который был использован для запуска контейнера. Показывает какие файлы были добавлены, изменены или удалены с момента его создания.

# docker events

**–** используется для просмотра событий, происходящих в Docker среде в реальном времени. Позволяет отслеживать создание, запуск, остановку и удаление контейнеров и др.

# docker inspect

**–** команда для получения подробной информации о Docker-объектах, таких как контейнеры, образы, сети и тома. Вернет Json объект с информацией об указанном объекте, такой как: идентификатор, имя, время создания, конфигурация, сетевые настройки, монтированные тома и др.

# docker port

**–** список проброшенных портов для заданного контейнера.

# docker top

**–** предоставляет информацию о процессах, выполняющихся внутри заданного контейнера. В действительности эта команда запускает утилиту Unix **ps** на хосте, и выбирает для вас процессы, выполняющиеся в заданном контейнере.

# docker build

**–** используется для создания образа на основе Dockerfile. Созданный образ в свою очередь может быть использован для запуска контейнеров.

# docker create

**–** создает контейнер из заданного образа, но не запускает его. Аргументы команды в основном те же, что для команды *docker run*.

# docker start

**–** запустить созданный, или остановленный контейнер.

# docker commit

**–** создает образ из указанного контейнера.

# docker export

**–** экспортирует содержимое файловой системы контейнера в виде tar-архива. Следует отметить, что все метаданные, такие как объявленные порты, команда CMD, ENTRYPOINT будут потеряны. В экспортируемую файловую систему не включаются какие-либо тома.

# docker import

**–** создает образ из архивного файла, содержащего файловую систему, созданного командой *docker export*. Архив может быть задан путем к файлу или в форме URL, а также передан в стандартный поток ввода STDIN.

# docker save

**–** сохраняет образы или репозитории в tar-архив. Образы можно задавать по идентификаторам или в форме *repository:tag*. Если задано только имя репозитория, то в архив будут сохранены все образы из этого репозитория.

# docker load

**–** загружает репозиторий из архива. Репозиторий может содержать несколько образов и тегов. В отличии от команды *docker import*, в загружаемые образы включены метаданные и история.

# docker tag

**–** связывает имя репозитория и тега с заданным образом.

# docker history

**–** выводит информацию о каждом уровне в образе.

# docker images

**–** выводит список локальных образов.

# docker rmi

**–** удаляет заданный образ или несколько образов.

# Образ Docker

**–** файл или шаблон для запуска контейнера. Он включает в себя исполняемые файлы, зависимости, конфигурацию и другие компоненты. Образы являются неизменяемыми и версионируются. После изменения образа создается его новая версия. Это позволяет управлять версиями приложений, и легко воспроизводить окружение в любой момент времени.

**Контекст создания образа –** набор локальных файлов и каталогов, к которым можно обращаться из инструкций ADD или COPY в Dockerfile (может быть и пустым). Указывается как аргумент команды build.

Эти файлы и каталоги передаются в демон Docker как часть процесса создания. Чтобы не получить задержек, не добавляйте в контекст большие каталоги с кучей файлов.

В качестве контекста можно указывать git-репозиторий.



По умолчанию Docker ищет файл Dockerfile в корневом каталоге контекста.

-**f –** указать расположение Dockerfile в контексте.



**.dockerignore –** файл работает аналогично .gitignore. Все файлы, указанные в .dockerignore будут проигнорированы при передаче контекста.

# Dockerfile

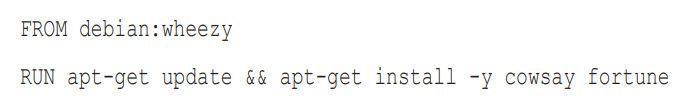
**–** текстовый файл, содержащий набор операций, которые используются для создания Docker-образа.

Инструкции:

**FROM образ –** определяет базовый образ ОС. Обязательна для Dockerfile.

**RUN –** определяет команду, выполняемую в командной оболочке **во время сборки** образа. Может быть использована для установки зависимостей, выполнения сценариев, копирования файлов.

Каждая инструкция RUN создает новый слой в образе, поэтому рекомендуется объединять их в одну строку с помощью &&.



**ENTRY-POINT –** позволяет определить выполняемый файл или команду, которая будет вызываться при запуске контейнера. В эту выполняемую программу передаются как аргументы любые инструкции *CMD* и аргументы команды *docker run*, записанные после имени образа.



**CMD –** Определяет аргументы по умолчанию для ENTRYPOINT или указывает команду, которая будет выполняться при запуске контейнера, если не определен ENTRYPOINT. CMD может быть переопределен при запуске контейнера с помощью *docker run* с аргументами.

**COPY –** скопировать файл из файловой системы хоста (вашего компьютера) в цифровую систему образа. Первый аргумент определяет файл на вашем компьютере, а второй – целевой путь. Можно использовать шаблонные символы для копирования нескольких фалов или каталогов. Нельзя копировать файлы, находящиеся вне контекста создания.

**ADD –** как и COPY может копировать файл(директорию) из контекста в образ. Помимо этого может автоматически распаковывать архивы и копировать файлы из удаленных URL.

**ENV –** определяет переменные среды внутри образа. На эти переменные можно ссылаться в следующих инструкциях Dockerfile, или во время работы контейнера.



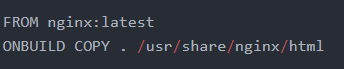
**EXPOSE –** используется для указания портов, которые контейнер будет слушать во время его выполнения. Это необходимо для того, чтобы внешние сервисы или другие контейнеры могли установить соединение с приложением, работающим внутри контейнера.

EXPOSE не выполняет фактического проброса портов, для этого необходимо использовать опцию **-p** при запуске контейнера с помощью *docker run*.

**MAINTAINER –** информация об авторе образа (на случай если заходите загрузить образ в реестр). Эта информация будет доступна в выводе команды *docker inspect*.



**ONBUILD –** используется для определения команд, которые будут выполнены автоматически наследующим образом, во время его сборки. Например, вы можете использовать ONBUILD, чтобы автоматически установить зависимости или скопировать файлы в определенные директории.



ONBUILD не выполняет никаких действий при обработке текущего Dockerfile.

**USER –** используется для указания пользователя, под которым будут выполняться команды внутри контейнера. Может указываться имя пользователя или идентификатор UID. По умолчанию выполняется под root.  
Таким образом мы можем настроить права пользователю и избежать выполнения всех команд под root.

**VOLUME –** указать путь, куда будет смонтирован volume. Если такой файл или каталог уже существует на хосте, то он копируется в том при запуске контейнера.

Фактическое монтирование выполняется при запуске контейнера используя опцию **-v** или **-mount** при использовании команды docker run.

**WORKDIR –** определяет рабочий каталог для всех последующих инструкций. То есть все команды далее будут выполнены в этом каталоге. Эту инструкцию можно использовать несколько раз. При использовании относительного пути он будет определятся относительно текущего WORKDIR.

**#** - комментарий

В некоторых инструкциях (RUN, CMD, ENTRYPOINT) допускается использование как формата командной оболочки так и формата *exec*.

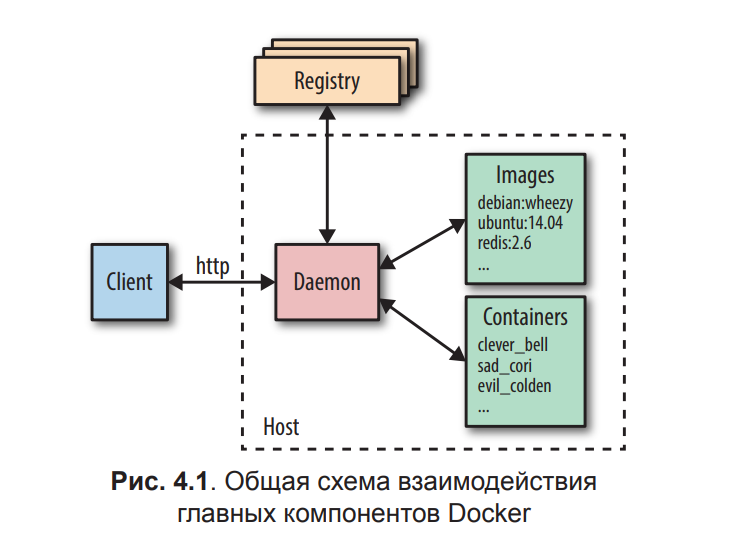
Формат **командной оболочки** – строка произвольной формы, передаваемае интерпретатору /bin/sh.

Формат **exec** принимает JSON-массив, предполагая, что первый элемент массива является именем исполняемого файла, а остальные элементы – параметры. например, *["executable", "param1", "param2"]*.

Используйте формат exec чтобы избежать случайного искажения строк командной оболочкой, или когда образ не содержит bin/sh.

# Архитектура Docker

* **Docker daemon (**демон Docker**) –** ответственный за создание, запуск и контроль работы контейнеров, и за создание и хранение образов. Запускается командой **docker daemon**, обычно его запускает операционная система.
* **Клиент Docker –** используется для взаимодействия с демоном по протоколу HTTP. Можно без затруднений организовать соединение с удаленными демонами Docker
* **Реестры Docker –** используются для хранения и распространения образов.



# Файловая система

Docker использует Union File System. Такие файловые системы позволяют подключать несколько файловых систем с наложением друг на друга, причем для пользователя они будут выглядеть как одна файловая система.

Docker поддерживает несколько различных реализаций: UnionFS, включая AUFS, Overlay, devicemapper, BTRFS и ZFS. Реализацию можно посмотреть с помощью команды **docker info** в заголовке **«Storage Driver»**. Драйвер файловой системы можно заменить, но это рекомендуется только в тех случаях, когда вы точно знаете, что делаете.

Для смены драйвера файловой системы перезапустите демон Docker передавая ему параметр:

**-s, --storage-driver** – установить драйвер файловой системы.

Чтобы сделать изменение постоянным, нужно отредактировать скрипт запуска или файл конфигурации Docker.

При смене драйвера файловой системы вы потеряете доступ ко старым контейнерам и образом. Восстановить доступ можно вернувшись к старой файловой системе. Чтобы перенести образ на новый драйвер просто сохраните нужный образ в tar-файл, затем распакуйте этот архив в новой файловой системе.

Образы Докера состоят из нескольких уровней. Каждый уровень представляет собой защищенную от записи файловую систему. Для каждой инструкции в Dockerfile создается свой уровень, который размещается поверх предыдущих уровней. Во время преобразования образа в контейнер (командой docker run или docker create) механизм Docker выбирает нужный образ и добавляет на самом верхнем уровне файловую систему с возможностью записи (одновременно с этим инициализируются разнообразные параметры настройки, такие как IP-адрес, имя, идентификатор и ограничения ресурсов).

**docker history <образ> -** позволяет увидеть набор уровней, формирующих образ.

Поскольку ненужные уровни значительно увеличивают размеры образов (а для файловой системы AUFS установлен строгий лимит, равный 127 уровням), во многих файлах Dockerfile можно обнаружить попытку свести к минимуму количество уровней посредством записи нескольких команд Unix в одной инструкции RUN.

Для **ускорения создания образов** Docker выполняет кэширование каждого уровня. Это позволяет переиспользовать уже созданные и неизмененные слои образа. Инструкции берутся из кэша при соблюдении условий:

* В кэше была обнаружена предыдущая инструкция;
* В кэше имеется уровень, который имеет в точности ту же инструкцию и предшествующий родительский уровень.

docker build **--no-cache –** запретить кэширование.

# ТОМА (VOLUMES)

**volumes –** это файлы или каталоги, которые монтируются непосредственно на хост (наш комп).

Тома используются для:

* Сохранение данных между запусками контейнера. Полезно для сохранения состояния БД и файлов конфигурации приложения при перезапуске контейнера.
* Обмен данными между контейнерами. Несколько контейнеров могут иметь доступ к общим данным.
* Получение в контейнере доступа к фалам с хоста.

Запустить контейнер с volume:



Указывается в формате **папка на хосте: папка в контейнере.**

В этом примере все, что контейнер запишет в указанный каталог, попадет в нашу папку на хостовом компьютере.

**docker volume create <volumeName> -** создать именованный volume. Мы можем управлять именованными томами и использовать их в контейнерах.



**--volumes-from <container> -** использовать тома другого контейнера. Таким образом несколько контейнеров смогут обмениваться данными между томами. Этот способ работает вне зависимости от того, активен ли в текущий момент контейнер, содержащий тома.

Том невозможно удалить, пока существует хотя бы один контейнер, установивший связь с этим томом.

# Связи между контейнерами

**Соединения (links) –** простейший способ обеспечения обмена информацией между контейнерами на одном хосте. Обмен данными будет происходить во внутренней сети Docker, то есть они останутся невидимыми из сети хоста.

**--link CONTAINER:ALIAS** в команде *docker run* – инициализирует соединение. *CONTAINER* – имя контейнера к которому мы хотим подключиться, *ALIAS* – локальное имя, которое мы используем внутри запускаемого контейнера для обращения к внешнему.



При этом имя и идентификатор контейнера будут добавлены в */etc/hosts* в управляющем контейнере.

Основным недостатком такого соединения является статичность. Несмотря на то, что при перезапуске контейнеров соединения должны сохраняться, они не обновляются, если контейнер-адресат заменен. Кроме того, контейнер адресат должен быть обязательно инициализирован раньше управляющего контейнера, то есть двунаправленное соединение установить невозможно.

Вообще, этот метод устарел и не рекомендуется к использованию в новых проектах.

Более продвинутыми являются соединения типа **network**. Доступны 4 основных режима:

* **bridge (мост)** – режим по умолчанию. Контейнеры могут взаимодействовать с другими контейнерами, подключенными к мосту. По умолчанию все контейнеры на хосте будут подключены к одной мостовой сети, что может быть нежелательно.

Можно создавать свои собственные мосты, и подключать к ним контейнеры. Внутри пользовательской сети смогут взаимодействовать только добавленные в сеть контейнеры.

Каждый контейнер будет иметь свой собственный внутренний IP. Docker автоматически настраивает DNS-сервер, чтобы контейнеры могли обращаться друг к другу по именами контейнеров.

Контейнеры также используют NAT(Network Address Translation) для связи с внешними ресурсами, такими как интернет и контейнеры в других сетях. Docker переводит внутренний IP на внешний IP-адрес хоста при попытке доступа ко внешним ресурсам.

* **host (хост) –** контейнер использует сетевые настройки хоста. Контейнер использует IP хоста и его ресурсы напрямую.  
  Контейнеры работающие в режиме host не изолированы друг от друга, могут обращаться напрямую друг к другу напрямую через localhost.
* **overlay** – предназначен для соединения нескольких контейнеров находящихся на разных Docker демонах.
* **macvlan** – с помощью него можно назначить MAC-адрес контейнеру, в результате чего он становится виден в сети как физическое устройство.
* **none –** контейнер полностью изолирован. Нет подключения. Взаимодействие с другими контейнерами невозможно.

**docker network create <networkName> -** создать сеть. Добавить контейнер в сеть можно при запуске с опцией *–network*.

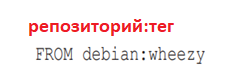
**docker network ls –** список активных сетей и их идентификаторы.

# Работа с реестрами

Есть официальный реестр Docker Hub, в котором хранятся различные образы докера. А есть и частные реестры. Вы можете использовать образы из реестров в качестве каркаса для своих реестров.

Образы хранятся в следующей иерархии:

* **Реестр –** сервис отвечающий за хранение и распространение образов. По умолчанию используется Docker Hub.
* **Репозиторий –** набор взаимосвязанных образов (обычно представляющих различные версии одного приложения или сервиса)
* **Тег –** алфавитно-цифровой идентификатор, присваиваемый образам внутри репозитория.



Если не указывать тег явно, то будет использован тег **latest.** Но лучше использовать явный тег.

**docker pull <образ> -** загрузить образ из репозитория на локальную машину.

**docker push –** загрузить свой образ на Docker Hub. Предварительно нужно создать учетную запись в реестре.

**docker search –** выводит список общедоступных репозиториев из реестра Docker Hub, соответствующих заданному шаблону поиска.

# DOCKER COMPOSE

**Docker Compose –** инструмент для создания и выполнения приложений, скомпонованных из нескольких Docker-контейнеров. Используется в основном при разработке и тестировании. Позволяет вам описывать структуру приложения в файле YAML и запускать с помощью одной команды.

**docker-compose up –** запускает контейнеры, описанные в файле docker-compose.yml.

-**d –** запустить контейнеры в фоновом режиме. По умолчанию мы будем видеть вывод логов, объединенный из всех контейнеров в один поток.

**docker-compose build** – пересоздание всех образов, созданных из файлов Dockerfile. Команда *up* будет создавать образы, только если они не существовали ранее. Поэтому build нужно использовать при необходимости обновления образов.

**docker-compose ps –** вывод информации о состоянии контейнеров, управляемых Compose.

**docker-compose run<service\_name> <command> -** используется для разового запуска контейнера, и выполнения команды внутри сервиса, определенного в файла docker-compose.yml. После выполнения команды контейнер будет остановлен и удален.

**docker-compose logs –** вывод журнальных записей с цветной подсветкой, объединенный для всех контейнеров, управляемых compose.

**docker-compose stop –** остановка контейнеров без их удаления.

**docker-compose rm –** удаление остановленных контейнеров.

-**v –** позволяет удалить все тома.

-**s –** позволяет также удалить все сети.

-**f –** удалить и сети, и тома.

Пример compose-файла:

version: '2'  
services:  
 artassessment-app:  
 image: artassessment  
 container\_name: 'app'  
 environment:- SPRING\_RABBITMQ\_HOST=rabbitmq  
 - SPRING\_RABBITMQ\_PORT=5672  
 - SPRING\_RABBITMQ\_USERNAME=admin  
 - SPRING\_RABBITMQ\_PASSWORD=admin  
 - SPRING\_RABBITMQ\_QUEUE=mail-queue  
 - SPRING\_RABBITMQ\_EXCHANGE=mail-exchange  
 - SPRING\_RABBITMQ\_ROUTEKEY=mail-routeKey  
 ports:  
 - 8080:8080  
 - 8000:8000  
 artassessment-postgresql:  
 container\_name: 'db'  
 extends:  
 file: postgresql.yml  
 service: artassessment-postgresql  
 keycloak:  
 container\_name: 'keycloak'  
 extends:  
 file: keycloak.yml  
 service: keycloak  
 rabbitmq:  
 container\_name: 'rabbitmq'  
 extends:  
 file: rabbitmq.yml  
 service: rabbitmq

**services –** список создаваемых контейнеров. Далее идет список сервисов: *rtassessment-app*, *artassessment-postgresql*, *keycloak*, *rabbitmq*.

**image –** образ, который используется для запуска контейнера. Здесь мы используем наш локальный образ, собранный командой docker build. Например для бд Postgresql: *image: postgres:10.4*.

**container\_name** – имя контейнера.

**environment** – список переменных окружения и их значений. Можно взять из файла с помощью опции **env-file**

**ports –** список пробрасываемых портов.

**extends –** используется для наследования конфигурации из другого файла docker-compose.yml.

**file –** файл из которого наследовать конфигурацию.

**service –** имя сервиса, который мы хотим экспортировать из другого файла.

Еще один пример docker-compose для PostgreSQL (всю эту инфу можно было бы объявить и в первом файле, вместо extends):

services:  
 artassessment-postgresql:  
 container\_name: "postgres"  
 build: postgres/  
 volumes:  
 - postgres-volume:/var/lib/postgresql/data  
 restart: unless-stopped  
 networks:  
 - mynet  
 ports:  
 - "5432:5432"  
 env\_file:  
 - .env

networks:  
 mynet:  
 driver: bridge  
  
volumes:  
 postgres-volume:

**build –** указывает путь к папке, в которой хранится Dockerfile для создания образа и запуска контейнера. В простейших случаях для бд может и не быть Dockerfile, тогда образ мы указываем в **image**.

**volumes** – список монтируемых томов. Здесь мы еще задаем ему имя. Это даст нам возможность использовать еще в каком-либо контейнере.

**restart –** настройка перезапуска контейнера в случае возникновения ошибки.

**networks –** список сетей. Сети используются для установления связи между контейнерами.

**driver –** указывает, какой драйвер используется для установки соединения между контейнерами.

# ОРКЕСТРАЦИЯ И КЛАСТЕРИЗАЦИЯ

**Оркестровка и управление кластером –** при развертывании большого количества контейнеров важно наличие инструментов для контроля и управления всей системой в целом. Каждый новый контейнер должен быть размещен на некотором хосте, его нужно контролировать и обновлять. Система должна правильно реагировать на сбои или изменения нагрузки, перемещая, запуская или останавливая контейнеры.

Основные решения: Kubernetes от Google, Marathon, Fleet от CoreOS (Операционная система, специализирующаяся на запуске контейнеров), Docker Swarm – собственный инструмент Docker.

**Кластеризация –** объединение виртуальных машин или аппаратных в единую систему. В кластере компьютеры(контейнеры) работают как единое целое, обмениваясь информацией и выполняя задачи.

Кластеризация позволяет увеличить производительность, отказоустойчивость и масштабируемость системы, распределяя нагрузку и обеспечивая резервирование ресурсов.

**Оркестрация –** обеспечение совместной работы всех элементов системы. Запуск контейнеров на соответствующих хостах и установление соединений между ними. Может также включать поддержку масштабирования, автоматического восстановления после критических сбоев и инструменты изменения балансировки нагрузки на узлы.

**Возможности оркестраторов:**

* **Управление масштабированием –** позволяют масштабировать приложения, управляя количеством запущенных контейнеров. Автоматически масштабируют приложение в зависимости от нагрузки, что позволяет эффективно использовать ресурсы и обеспечивает высокую производительность.
* **Отказоустойчивость.** Оркестраторы позволяют запускать несколько копий контейнеров на разных узлах. Если один узел выходит из строя, оркестратор автоматически переносит работу на другие доступные узлы, обеспечивая непрерывную работу приложения.
* **Сетевая организация.** Оркестраторы предоставляют функциональность для настройки сетевых взаимодействий между контейнерами. Позволяют создавать внутренние сети для изолирования контейнеров и обеспечивают возможность связи между контейнерами и внешними системами.
* **Управление конфигурацией.** Оркестраторы позволяют управлять конфигурацией и параметрами запуска контейнеров с помощью файлов конфигурации и инструментов командной строки.
* **Мониторинг и логирование.** Оркестраторы предоставляют возможности для мониторинга и сбора журналов работы контейнеров.

**Swarm –** собственный инструмент Docker для кластеризации контейнеров.

На каждом хосте запускается **агент (agent)** Swarm, а на отдельном хосте запускается **менеджер (manager)**.

**Менеджер** отвечает за оркестрацию и планирование работы контейнеров, распределенных по хостам.

Имеется несколько механизмов **обнаружения** хостов и добавления их в кластер. По умолчанию используется **обнаружение на основе токена** (token based discovery), при котором адреса хостов содержатся в списке, хранящемся в реестре Docker Hub.

**docker run swarm create –** создать токен для кластера.

При выполнении запуска контейнера, Swarm перехватит наш запрос, и перенаправит команду на наиболее подходящий хост. Таким образом контейнер будет создан на одном из хостов кластера.

**Kubernates –** средство оркестрации контейнеров, созданное компанией Google**.**