

Лабораторный практикум по Docker

выполняются в ОС Linux

Лабораторная работа 1. Hello World

Для начала найдем интересующий нас образ (первая команда) и установим его:

```
docker search redis  
docker run -d redis
```

Для запуска контейнера в фоновом режиме необходимо указать параметр `-d`. По умолчанию Docker запускает последнюю доступную версию. Если требуется определенная версия, она может быть указана в виде тега, например, версия 3.2 будет `docker run -d redis: 3.2`.

Для просмотра информации о запущенных в фоне контейнерах есть команда:

```
docker ps
```

Так же имеется две полезные команды. Первая выводит дополнительную информацию о контейнере. Вторая выводит его логи.

```
docker inspect <friendly-name|container-id>  
docker logs <friendly-name|container-id>
```

Redis установлен, но не доступен вне контейнера.

6379 — используемый порт Redis. Для проброса порта имеется такая команда **`-p <host-port>:<container-port>`** . Пример:

```
docker run -d --name redisHostPort -p 6379:6379 redis:latest
```

По умолчанию порт на хосте сопоставлен с 0.0.0.0, что означает доступ со всех IP-адресов. Вы можете указать конкретный IP-адрес при определении сопоставления портов, например, **`-p 127.0.0.1:6379:6379`**

Как хранить данные и не потерять их при переустановке контейнера

Если открыть [документацию по Redis для Docker](#), то увидим информацию по хранению данных. Данный образ хранит её в /data .

Любые данные, которые необходимо сохранить на Docker хосте, а не внутри контейнера redis, должны храниться в /opt/docker/data/redis, что задается отдельным параметром:

```
docker run -d --name redisMapped -v /opt/docker/data/redis:/data  
redis
```

Взаимодействие внутри контейнера

Ранее мы использовали -d для выполнения контейнера в отдельном фоновом состоянии. Без указания этого контейнер будет работать на переднем плане.

Если нужно взаимодействовать с контейнером (например, для доступа к оболочке bash), нужно добавить опции -it.

Примеры:

```
docker run ubuntu ps запустит контейнер ubuntu и выполнит команду ps.
```

```
docker run -it ubuntu bash даст доступ к bash shell внутри контейнера.
```

Лабораторная работа 2. HTML страница

Цель урока: создать веб сайт со статичной страницей в контейнере.

Создадим Dockerfile

Docker images начинаются с базового image(образа), который включает в себя зависимости платформы для приложения. Этот базовый образ определяется как команда в **Dockerfile**, который является списком инструкций (команд), описывающих как развернуть приложение.

В примерах будет использован как базовый образ NGINX версии Alpine (настроенный веб сервер в дистрибутиве linux alpine).

Так же создайте простой index.html файл в папке, из которой идет работа.

Создадим Dockerfile:

```
FROM nginx:alpine
COPY . /usr/share/nginx/html
```

Первая строчка определяет базовый образ. Вторая строчка копирует контент текущей папки во внутрь контейнера (наш index.html).

Сборка Docker Image

Для сборки используется команда build. Она принимает несколько параметров. Например, параметр **-t** позволяет указать имя для и тег для изображения (используется часто как номер версии). Пример:

```
docker build -t webserver-image:v1 .
```

Для просмотра списка изображений используйте команду:

```
docker images
```

Время запускать!

Тут ничего нового в сравнении с прошлым уроком. Запускаем контейнер с пробросом 80 порта.

```
docker run -d -p 80:80 webserver-image:v1
```

Теперь можно проверить — работает ли наш сайт? Воспользуемся утилитой curl.

```
curl docker
```

Лабораторная работа 3. Сборка образов

В предыдущих работах речь была о запуске контейнеров из уже существующих образов Docker. Тут будет описано как собрать образ, основываясь на собственных требованиях.

Для этого занятия контейнер будет запускать статическое HTML-приложение с использованием веб-сервера Nginx.

Имя компьютера, на котором запущен контейнер: Docker. Если вы хотите получить доступ к какой-либо из служб — используйте docker вместо localhost или 0.0.0.0.

Об образах Docker

Образы Docker создаются на основе Dockerfile. Dockerfile определяет все шаги, необходимые для создания образа Docker с приложением, настроенным и готовым к запуску в качестве контейнера. Сам образ содержит все, от операционной системы до зависимостей и конфигурации, необходимых для запуска приложения.

Образ позволяет переносить его между различными средами и быть уверенным, что он заработает в любой из них.

Dockerfile позволяет пользователям расширять существующие изображения вместо создания с нуля. Основываясь на существующем образе, нужно только определить шаги по настройке приложения. Базовые образы могут быть основными установками операционной системы или настроенными системами, которые просто нуждаются в дополнительных настройках.

Начнем создавать

Все образы Docker начинаются с **базового образа**. Базовый образ — это те же изображения из реестра Docker, которые используются для запуска контейнеров. Наряду с именем образа мы также можем включить тег, чтобы указать, какую конкретную версию мы хотим, по умолчанию это последняя версия.

Эти базовые образы используются в качестве основы для запуска вашего приложения. Например, в этом уроке мы требуем, чтобы NGINX был настроен и запущен в системе, прежде чем мы сможем развернуть наши статические HTML-файлы. Поэтому мы хотим использовать NGINX в качестве базового образа.

Dockerfile — это простые текстовые файлы с командой в каждой строке. Чтобы определить базовый образ, нужно использовать инструкцию **FROM <image-name>: <tag>**. Добавим образ в Dockerfile.

```
FROM nginx:1.11-alpine
```

Важно: заманчиво использовать тег **:latest**, однако есть вероятность построить образ не на той версии, которую бы хотелось. Для исключения ошибок в работе приложения рекомендуется использовать конкретную версию.

Время команд в Dockerfile

Определив базовый образ, нам нужно запустить различные команды для настройки нашего образа. Есть много команд, которые могут помочь с этим, две главные команды это **COPY** и **RUN**.

RUN <команда> позволяет вам выполнить любую команду, как в командной строке, например, установить различные пакеты приложений или выполнить команду сборки. Результаты RUN сохраняются в образе, поэтому важно не оставлять ненужные или временные файлы на диске, так как они будут включены в образ.

COPY <src> <dest> позволяет копировать файлы из каталога, содержащего Dockerfile, в образ контейнера. Это чрезвычайно полезно для исходного кода и ресурсов, которые вы хотите развернуть внутри своего контейнера.

Для примера создадим новый файл index.html, который добавим в контейнер. На следующей строке после команды FROM созданного ранее dockerfile добавим команду команду COPY, чтобы скопировать index.html в каталог с именем /usr/share/nginx/html

```
COPY index.html /usr/share/nginx/html/index.html
```

С настроенным образом Docker и определившись, какие порты будут открыты, нужно указать команду, которая запускает приложение.

С настроенным образом Docker и определением, какие порты мы хотим сделать доступными, теперь нам нужно определить команду, которая запускает приложение.

Строка CMD в Dockerfile определяет команду по умолчанию, запускаемую при запуске контейнера. Если команде требуются аргументы, рекомендуется использовать массив, например:

```
["cmd", "-a", "arga value", "-b", "argb-value"]
```

Массив будет объединен вместе и итоговая команда ниже будет выполнена:

```
cmd -a "arga value" -b argb-value
```

Ещё пример — передадим команду nginx. По-умолчанию демон NGINX будет выключен:

```
CMD ["nginx", "-g", "daemon off;"]
```

Порты

Файлы скопированы в наш образ, теперь нужно определить, на каком порте приложение должно быть доступно.

Используя команду **EXPOSE <port>**, нужно сообщить Docker, какие порты должны быть открыты. Вы можете определить несколько портов в одной команде, например, **EXPOSE 80 433** или **EXPOSE 7000-8000**. Добавим в наш файл конфигурации:

```
EXPOSE 80
```

Время собрать контейнер

После написания Dockerfile нужно использовать команду **docker build**, чтобы превратить его в образ. Команда build принимает каталог, содержащий Dockerfile, выполняет шаги и сохраняет образ в вашем локальном Docker Engine. Если произошел сбой из-за ошибки, сборка прекращается.

Итоговый dockerfile приведен ниже. Это nginx версии 1.11, в который скопирована html страница и выключен демон веб-службы.

```
FROM nginx:1.11-alpine COPY index.html  
/usr/share/nginx/html/index.html EXPOSE 80 CMD ["nginx", "-g",  
"daemon off;"]
```

Итак, выполняем команду **docker build** для создания образа. Можно дать образу понятное имя, используя опцию **-t <name>**.

```
docker build -t my-nginx-image:latest .
```

Не стоит забывать о команде **docker images** для просмотра списка образов.

Время запуска

После успешного создания образа можно запустить контейнер так же, как это описано в первой лабораторной работе.

NGINX предназначен для работы в качестве фоновой службы, поэтому ему нужно включить опцию **-d**. Чтобы сделать веб-сервер доступным, нужно привязать его к порту 80, используя **-p 80:80**. Например:

```
docker run -d -p 80:80 image-id или friendly-tag-name
```

Теперь можно получить доступ к запущенному веб-серверу по имени хоста `docker`.

После запуска контейнера команда **`curl -i http: // docker`** вернет наш `index.html` через NGINX и созданный нами образ.

Лабораторная работа 4. Сборка приложения node.js

В этой работе продолжается изучение того, как создавать и развертывать приложения в виде контейнера Docker. Предыдущий сценарий охватывал развертывание статического HTML-сайта. В этой работе рассматривается, как развернуть приложение Node.js в контейнере.

Среда настроена с доступом к личному экземпляру Docker, а код для приложения Expressjs по умолчанию находится в рабочем каталоге.

Имя компьютера, на котором запущен Docker, называется Docker. Если нужно получить доступ к какой-либо службе, следует использовать hostname docker вместо localhost или 0.0.0.0.

Базовый образ

Как описано в предыдущем занятии, все образы начинаются с базового изображения, в идеале максимально приближенного к желаемой конфигурации. Для Node.js есть готовые образы с тегами для каждой выпущенной версии.

Образ для node 10.0 это **node:10-alpine**. Это образ на основе alpine linux, который меньше и более обтекаема, чем официальное изображение.

Помимо базового образа также необходимо создать базовые каталоги, из которых запускается приложение. Используя **RUN <command>**, можно выполнять команды, как если бы они выполнялись из командной оболочки. Используя **mkdir**, можно создавать каталоги, из которых будет выполняться приложение. Сейчас идеальным каталогом был бы **/src/app**, поскольку пользователь среды имеет доступ для чтения/записи к этому каталогу.

Можно определить рабочий каталог, используя **WORKDIR <каталог>**, чтобы гарантировать, что все будущие команды будут выполняться из каталога нашего приложения.

В нашем Dockerfile определим выше описанное:

```
FROM node:10-alpine
RUN mkdir -p /src/app
WORKDIR /src/app
```

Работа с NPM

Базовый образ и местоположение приложения настроены. Следующим этапом является установка зависимостей, необходимых для запуска приложения. Для Node.js это означает запуск установки NPM.

Чтобы свести время сборки к минимуму, Docker кэширует результаты выполнения строки в файле Docker для использования в будущей сборке. Если что-то изменилось, то Docker сделает недействительными текущую и все последующие строки, чтобы убедиться, что все обновлено.

Сделаем так, чтобы `npm install` запускался только в том случае, если что-то в нашем файле `package.json` изменилось. Если ничего не изменилось, то продолжим использовать версию кеша для ускорения развертывания. Используя **COPY package.json <dest>**, можно сделать кеш команды **RUN npm install** недействительным, если файл `package.json` изменился. Если файл не изменился, то кэш не будет признан действительным, и будут использованы кэшированные результаты команды установки `npm`.

Добавим описанные выше действия в `dockerfile`:

```
COPY package.json /src/app/package.json
RUN npm install
```

Пример `package.json`:

```
{
  "name": "scrapbook-node-docker-client-as-container",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "express": "~4.9.0",
    "body-parser": "~1.8.1",
    "cookie-parser": "~1.3.3",
    "morgan": "~1.3.0",
    "serve-favicon": "~2.1.3",
    "debug": "~2.0.0",
    "jade": "~1.6.0"
  }
}
```

Это перевод-заметка для себя с сайта katacoda.com. Полные копии приложения есть там с возможностью запуска описанных тут команд.

Время развертывания

Создадим необходимые шаги в Dockerfile, чтобы завершить развертывание приложения, как это было на прошлом занятии.

Скопируем файлы для развертывания, в Dockerfile следует использовать **COPY**. **<dest dir>**.

После того, как исходный код скопирован, порты, к которым требуется доступ к приложению, определяются с помощью **EXPOSE <порт>**.

Наконец, приложение готово к установке зависимостей. При использовании Node.js следует использовать один хитрый прием: запустить команду npm. Параметр start описан в «scripts» — «start» файла package.json. Ниже приведен пример и краткое описание — почему так.

```
COPY . /src/app
EXPOSE 3000
CMD [ "npm", "start" ]
```

После того, как мы установили наши зависимости с помощью `npm install`, мы хотим скопировать остальную часть исходного кода нашего приложения. Разделение установки зависимостей и копирование исходного кода позволяет нам использовать кеш при необходимости.

Если мы скопируем наш код перед запуском `npm install`, он будет запускаться каждый раз, так как наш код изменился бы. Копируя просто `package.json`, мы можем быть уверены, что кеш становится недействительным только тогда, когда содержимое нашего пакета изменилось. Итоговый `dockerfile`:

```
FROM node:10-alpine
RUN mkdir -p /src/app
WORKDIR /src/app
COPY package.json /src/app/package.json
RUN npm install
COPY . /src/app
EXPOSE 3000
CMD [ "npm", "start" ]
```

Запускаем

```
docker build -t my-nodejs-app .
docker run -d --name my-running-app -p 3000:3000 my-nodejs-app
```

И проверяем:

```
curl http://docker:3000
```

Переменные окружения

Образы `docker` должны быть спроектированы так, чтобы их можно было переносить из одной среды в другую без внесения каких-либо изменений или необходимости восстановления. Следуя этому шаблону, вы можете быть уверены, что если он

работает в одной среде, например, в промежуточной, то он будет работать в другой среде, например в рабочей среде.

С помощью Docker переменные среды могут быть определены при запуске контейнера. Например, для приложений Node.js вы должны определить переменную среды для NODE_ENV при запуске.

Используя опцию -e, вы можете установить имя и значение **-e NODE_ENV = production**

```
docker run -d --name my-production-running-app -e  
NODE_ENV=production -p 3000:3000 my-nodejs-app
```

Лабораторная работа 5. Docker OnBuild

В этом сценарии мы рассмотрим, как можно оптимизировать Dockerfile, используя OnBuild инструкции.

Среда настроена с помощью примера приложения Node.js (предыдущая лабораторная работа 4), однако подходы могут быть применены к любому изображению. Имя компьютера, на котором запущен Docker, называется Docker. Если нужно получить доступ к какой-либо из служб, тогда используйте docker вместо localhost или 0.0.0.0.

Об OnBuild

В то время как Dockerfile выполняется в порядке сверху вниз, с помощью OnBuild вы можете запустить инструкцию, которая будет выполнена позже, когда образ используется в качестве основы для другого образа.

В результате вы можете отложить выполнение, чтобы оно зависело от приложения, которое вы создаете, например, от файла package.json приложения.

Ниже приведен файл Docker Node.js OnBuild. В отличие от сценария предыдущего занятия, команды приложения имеют префикс ONBUILD.

```
FROM node:7
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
ONBUILD COPY package.json /usr/src/app/
ONBUILD RUN npm install
ONBUILD COPY . /usr/src/app
CMD [ "npm", "start" ]
```


В результате образ будет построен, но команды приложения с пометкой ONBUILD не будут выполняться до тех пор, пока построенный образ не будет использован в качестве базового. Затем они будут выполнены как часть сборки базового образа.

Пример

Мы имеем всю логику для копирования кода, установки зависимостей и запуска приложения. Для примера добавим ещё один элемент — откроем порт.

Преимущество создания образов OnBuild состоит в том, что наш Dockerfile теперь намного проще и может легко использоваться повторно в нескольких проектах без необходимости повторного выполнения одних и тех же шагов, что сокращает время сборки.

Пример такого dockerfile:

```
FROM node:7-onbuild  
EXPOSE 3000
```

Сборка и запуск контейнера

Для начала — сборка первого контейнера из базового docker образа.

```
docker build -t my-nodejs-app .
```

Теперь выполним команду сборки встроенного образа, который основан на предыдущем:

```
docker run -d --name my-running-app -p 3000:3000 my-nodejs-app
```

Сборка первого контейнера заняла около 2 минут при наличии хорошего интернета. Сборка второго контейнера заняла пару секунд.

Как обычно можно проверить доступность сервиса в контейнере командой:

```
curl http://docker:3000
```

Лабораторная работа 6.

Игнорирование файлов при сборке

29 Сентября, 2020

В этой лабораторной работе мы рассмотрим, как можно игнорировать попадание определенных файлов в образ Docker, что может представлять угрозу безопасности и увеличивать время сборки.

Чтобы не допустить ошибочного включения в изображения конфиденциальных файлов или каталогов, нужно добавить файл с именем `.dockerignore`.

Пример

Dockerfile копирует рабочий каталог в образ Docker. В результате это может включать потенциально конфиденциальную информацию, такую как файл паролей, которым мы хотели бы управлять за пределами изображения. Добавим файл `passwords.txt` с нашим условным паролем, а так же напишем Dockerfile:

```
FROM alpine
ADD . /app
COPY cmd.sh /cmd.sh
CMD ["sh", "-c", "/cmd.sh"]
```

Соберем образ и посмотрим файлы в нем (сюрприз — файл `passwords.txt` будет в контейнере):

```
docker build -t password .
docker run password ls /app
```

Как игнорировать файл?

Следующая команда включит `passwords.txt` в файл `.dockerignore` и гарантирует, что он случайно не окажется в контейнере.

```
echo passwords.txt >> .dockerignore
```

Файл игнорирования поддерживает каталоги и регулярные выражения для определения ограничений, очень похожих на .gitignore у git.

Этот файл также можно использовать для уменьшения времени сборки, что будет рассмотрено позже. Соберем образ, теперь файла с паролем быть не должно.

```
docker build -t nopassword .
```

Если вам нужно использовать пароли как часть команды RUN, вам нужно скопировать, выполнить и удалить файлы как часть одной команды RUN. Внутри образа сохраняется только конечное состояние контейнера Docker.

Улучшаем время сборки

Файл .dockerignore может гарантировать, что конфиденциальные данные не будут включены в образ Docker. Однако его также можно использовать для улучшения времени сборки образа.

Для примера создадим файл в 100 МБ, например big-temp-file.img и соберем образ. Это уже займет намного больше времени, чем обычно.

```
docker build -t large-file-context .
```

Так же целесообразно игнорировать каталоги .git вместе с зависимостями, которые загружаются /создаются в образе, такими как node_modules. Они никогда не используются приложением, работающим в Docker-контейнере, а просто добавляют издержки процессу сборки.

Таким же образом мы использовали файл .dockerignore для исключения конфиденциальных файлов, мы можем использовать его для исключения файлов, которые мы не хотим отправлять в сборку образа Docker.

```
echo big-temp-file.img >> .dockerignore  
docker build -t no-large-file-context .
```

Лабораторная работа 7. Контейнеры данных

Контейнеры могут выполнять разные функции. Это может быть контейнер, создающий и хранящий данные, например, база данных. Так же это могут быть данные, требующие дополнительной конфигурации или SSL-сертификации.

Один из подходов, который был ранее, заключается в использовании опции `-v <host-dir>: <container-dir>` для сопоставления каталогов. Другой подход заключается в использовании контейнеров данных, о которых сегодня и пойдет речь.

Создадим контейнер

Контейнеры данных — это контейнеры, единственная цель которых заключается в том, чтобы быть местом для хранения/управления данными.

Как и другие контейнеры, они управляются хост-системой. Однако они не запускаются при выполнении команды **`docker ps`**.

Чтобы создать контейнер данных, мы сначала создаем контейнер с понятным именем для использования в будущем. Будем использовать `busybox` в качестве базы, так как он небольшой и легкий на тот случай, если мы захотим перенести контейнер на другой хост.

При создании контейнера мы также используем опцию `-v`, чтобы определить, где другие контейнеры будут считывать/сохранять данные.

```
docker create -v /config --name dataContainer busybox
```

Теперь, когда контейнер установлен, можно копировать файлы из локального клиентского каталога в контейнер.

Для копирования файлов в контейнер нужно использовать команду `docker cp`. Следующая команда скопирует файл `config.conf` в наш `dataContainer` и каталог `config`.

```
docker cp config.conf dataContainer:/config/
```

Монтирование томов

Теперь контейнер данных имеет конфиг файл и сейчас можно ссылаться на контейнер при запуске зависимых контейнеров, требующих этот файл конфигурации.

Используя опцию `—volumes-from <container>`, можно использовать тома монтирования из других контейнеров. В примере ниже запустим контейнер `Ubuntu`, который имеет ссылку на контейнер данных. Когда мы выведем каталог конфигурации, он покажет файлы из прикрепленного контейнера.

```
docker run --volumes-from dataContainer ubuntu ls /config
```

Если каталог `/config` уже существует, то имя тома будет переопределено. Так же можно привязать несколько томов.

Экспорт\импорт контейнеров

Если мы хотим переместить контейнер данных на другой компьютер, мы можем экспортировать его в файл `.tar`.

```
docker export dataContainer > dataContainer.tar
```

Для импорта контейнера данных назад в `Docker` используйте команду:

```
docker import dataContainer.tar
```

Лабораторная работа 8. Создание сетей между контейнерами с использованием Links

В этой лабораторной работе рассматривается, как разрешить нескольким контейнерам взаимодействовать друг с другом: как подключить хранилище данных (в данном случае Redis) к приложению, запущенному в отдельном контейнере.

Запустим redis

Наиболее распространенным сценарием подключения к контейнерам является приложение, подключающееся к хранилищу данных. Ключевым аспектом при создании link является имя контейнера.

Настроим:

```
docker run -d --name redis-server redis
```

Создадим Link

Для подключения к исходному контейнеру используется команда **—link <container-name|id>:<alias>** при запуске нового контейнера. container-name|id относится к исходному контейнеру, который определен ранее (redis), а alias определяет понятное имя хоста.

Устанавливая alias, мы разделяем, как наше приложение настроено на инфраструктуру. Это означает, что конфигурацию приложения не нужно менять, поскольку она подключена к другим средам.

Рассмотрим на примере. Вызовем контейнер Alpine, который связан с redis-сервером. Определим псевдоним как Redis. Когда link будет создан, Docker сделает две вещи.

Во-первых, Docker установит некоторые переменные окружения на основе ссылки на контейнер. Эти переменные среды дают возможность ссылаться на порты и IP-адреса, через известные имена.

Можно вывести все переменные окружения, добавив команду `env`. Например, создадим `link` и выведем `env`:

```
docker run --link redis-server:redis alpine env
```

Во-вторых, Docker обновит файл `HOSTS` контейнера с записью для исходного контейнера с тремя именами: оригинал, `alias` и `hash-id`. Можно вывести запись `host` контейнеров, используя `cat /etc/hosts`

```
docker run --link redis-server:redis alpine cat /etc/hosts
```

Пример вывода:

```
172.18.0.2 redis a2f3ba2b1d3c redis-server  
172.18.0.3 43106ddc9e26
```

После создания `link`, можно пропинговать исходный контейнер так же, как если бы это был сервер, работающий в сети.

```
docker run --link redis-server:redis alpine ping -c 1 redis
```

Подключение приложения

Создав `link`, приложения могут подключаться и взаимодействовать с исходным контейнером обычным способом, независимо от того, что обе службы работают в контейнерах.

ВАЖНО! Это перевод с сайта интерактивных уроков katacoda.com, все контейнеры с приложениями есть только там

Используем простое приложение на `node.js`, которое будет подключаться к `redis`:


```
docker run -d -p 3000:3000 --link redis-server:redis  
katakoda/redis-node-docker-example
```

В итоге получим работающее приложение, которое можно проверить http запросом:

```
curl docker:3000
```

Пример с redis cli

Таким же образом можно подключать контейнеры к приложениям и к своим собственным инструментам CLI.

Эта команда запустит инстанс redis-cli и подключится к серверу по alias:

```
docker run -it --link redis-server:redis redis redis-cli -h  
redis
```

Для проверки работы можно вывести все данные из redis командой:

```
KEYS *
```

Лабораторная работа 9. Создание сетей между контейнерами с использованием NETWORK

В этом занятии будет описано как создать сеть docker, которая позволяет контейнерам взаимодействовать по сети. Так же будет рассмотрен встроенный dns сервер.

У Docker есть два подхода к работе в сети. Первый — это рассмотренный в прошлой лекции link, который обновляет /etc/hosts и переменные среды, чтобы позволить контейнерам подключаться.

Альтернативный подход заключается в создании Docker-сети, к которой подключены контейнеры. Сеть имеет атрибуты, аналогичные физической сети.

Создание сети

Первым шагом будет создание сети с использованием CLI. Эта сеть позволит нам прикрепить несколько контейнеров, которые смогут подключаться друг к другу.

Создадим сеть с желаемым именем:

```
docker network create backend-network
```

Теперь при запуске контейнера можно использовать ключ **—net**, чтобы определить подключение к сети. Например:

```
docker run -d --name=redis --net=backend-network redis
```

Сетевое взаимодействие

В отличие от использования link, docker-сеть ведет себя как традиционные сети, где узлы могут быть подключены / отключены.

Первое отличие, это то, что Docker больше не назначает переменные среды и не обновляет файл `hosts` для контейнеров. Можно вывести две команды ниже и увидеть, что упоминаний других контейнеров нет.

```
docker run --net=backend-network alpine env  
docker run --net=backend-network alpine cat /etc/hosts
```

Вместо этого контейнеры могут связываться через встроенный DNS-сервер. Этот DNS-сервер назначается всем контейнерам как IP 127.0.0.11 и устанавливается в файле `resolv.conf`. Проверим это командой:

```
docker run --net=backend-network alpine cat /etc/resolv.conf
```

Когда контейнеры пытаются получить доступ к другим контейнерам через имя, такое как Redis, DNS-сервер возвращает IP-адрес контейнера. В этом случае полное имя Redis будет `redis.backend-network`.

```
docker run --net=backend-network alpine ping -c1 redis
```

Подключение двух контейнеров

Docker поддерживает одновременное подключение сетей и контейнеров к нескольким сетям.

Например, давайте создадим отдельную сеть с приложением Node.js, которое связывается с существующим экземпляром Redis. Создадим сеть:

```
docker network create frontend-network
```

Используем команду подключения существующего контейнера к сети:

```
docker network connect frontend-network redis
```

Если запустить тестовое приложение, то оно сможет обмениваться данными с контейнером `redis`.

```
docker run -d -p 3000:3000 --net=frontend-network  
katakoda/redis-node-docker-example
```

```
curl docker:3000
```

Aliases при использовании сети

Link по-прежнему поддерживаются при использовании Docker Network и предоставляют способ определения псевдонима для имени контейнера. Это даст контейнеру дополнительное имя DNS-записи и способ обнаружения. При использовании `—link` встроенный DNS гарантирует, что результат поиска будет только в том контейнере, где используется `—link`.

Другой подход заключается в предоставлении псевдонима при подключении контейнера к сети.

Следующая команда подключит экземпляр Redis к внешней сети с псевдонимом db:

```
docker network create frontend-network2  
docker network connect --alias db frontend-network2 redis
```

Когда контейнеры пытаются получить доступ к сервису через имя db, ими будет получен IP-адрес контейнера Redis.

Отключение от сети

Создав сети, можно использовать CLI для изучения деталей. Выведем список сетей.

```
docker network ls
```

Так же можно посмотреть какие контейнеры подключены и их IP.

```
docker network inspect frontend-network
```

А теперь отключим контейнер redis от сети frontend-network:

```
docker network disconnect frontend-network redis
```

Лабораторная работа 10. Сохранение данных с использованием томов (Volumes)

Docker Volumes позволяет обновлять контейнеры, перезагружать машины и обмениваться данными без потери данных. Это важно при обновлении базы данных или версий приложения.

Тома Docker создаются и назначаются при запуске контейнеров. Тома позволяют сопоставить каталог хоста с контейнером для обмена данными.

Тома являются двунаправленными. Это позволяет получать доступ к данным, хранящимся на хосте, из контейнера. Это также означает, что данные, сохраненные процессом внутри контейнера, сохраняются и на хосте.

Подключение тома

В этом примере Redis будет использоваться для сохранения данных. Запустим контейнер Redis и создадим том данных с помощью параметра `-v`. Это указывает, что любые данные, сохраненные в контейнере в каталоге `/data`, должны сохраняться на хосте в каталоге `/docker/redis-data`.

```
docker run -v /docker/redis-data:/data --name r1 -d redis  
\redis-server --appendonly yes
```

Запишем данные в redis командой:

```
cat data | docker exec -i r1 redis-cli --pipe
```

Если команда выполнена успешно, то в папке `/docker/redis-data` будет создан файл `aof`.

```
ls /docker/redis-data
```

Так же путь на хосте можно использовать для других контейнеров. Например, сделаем контейнер с ubuntu, который будет заниматься резервным копированием.

```
docker run -v /docker/redis-data:/backup ubuntu ls /backup
```

Shared Volumes

Тома, сопоставленные с хостом, отлично подходят для сохранения данных. Однако, чтобы получить доступ к ним из другого контейнера, вам нужно знать точный путь, что может привести к опечаткам и другим ошибкам.

Альтернативный подход заключается в использовании **—volumes-from**. Параметр отображает тома из исходного контейнера в запускаемый контейнер.

Теперь сопоставляем том нашего контейнера Redis с контейнером Ubuntu. Каталог /data существует только в контейнере Redis, однако с помощью -volumes-from контейнер Ubuntu может получить доступ к данным.

```
docker run --volumes-from r1 -it ubuntu ls /data
```

Read-only тома

Монтирование томов дает контейнеру полный доступ на чтение и запись к каталогу. Можно указать разрешения только для чтения каталога, добавив **ro** к монтированию.

ВАЖНО: если контейнер попытается изменить данные в каталоге, произойдет ошибка: Read-only file system.

```
docker run -v /docker/redis-data:/data:ro -it
```