

Docker

Пример "контейнеризованного" React/Vue/Express/Postgres приложения.

Введение, Docker CLI и Dockerfile

Что такое Docker?

- Начало работы с Docker на английском языке.
- Перевод этого "начала" на русский язык от Microsoft.

Docker - это открытая платформа для разработки, доставки и запуска приложений. Он позволяет отделить приложения от инфраструктуры и управлять инфраструктурой по аналогии с тем, как мы управляем приложениями.

Docker предоставляет возможность упаковывать и запускать приложение в слабо изолированной среде - контейнере. Изоляция и безопасность позволяют одновременно запускать несколько контейнеров на одном хосте (хостом может быть наша локальная машина, дата центр, облачный провайдер или их микс). Контейнеры являются легковесными и содержат все необходимое для запуска приложения, что избавляет нас от необходимости полагаться на то, что установлено на хосте.

Для чего Docker может использоваться?

Быстрая и согласованная доставка приложений

Docker рационализирует жизненный цикл разработки, позволяя разработчикам работать в стандартизированной среде через локальные контейнеры, предоставляющие приложения и сервисы. Контейнеры отлично подходят для рабочих процессов непрерывной интеграции и непрерывной доставки (continuous integration/continuous delivery, CI/CD).

Отзывчивая разработка и масштабирование

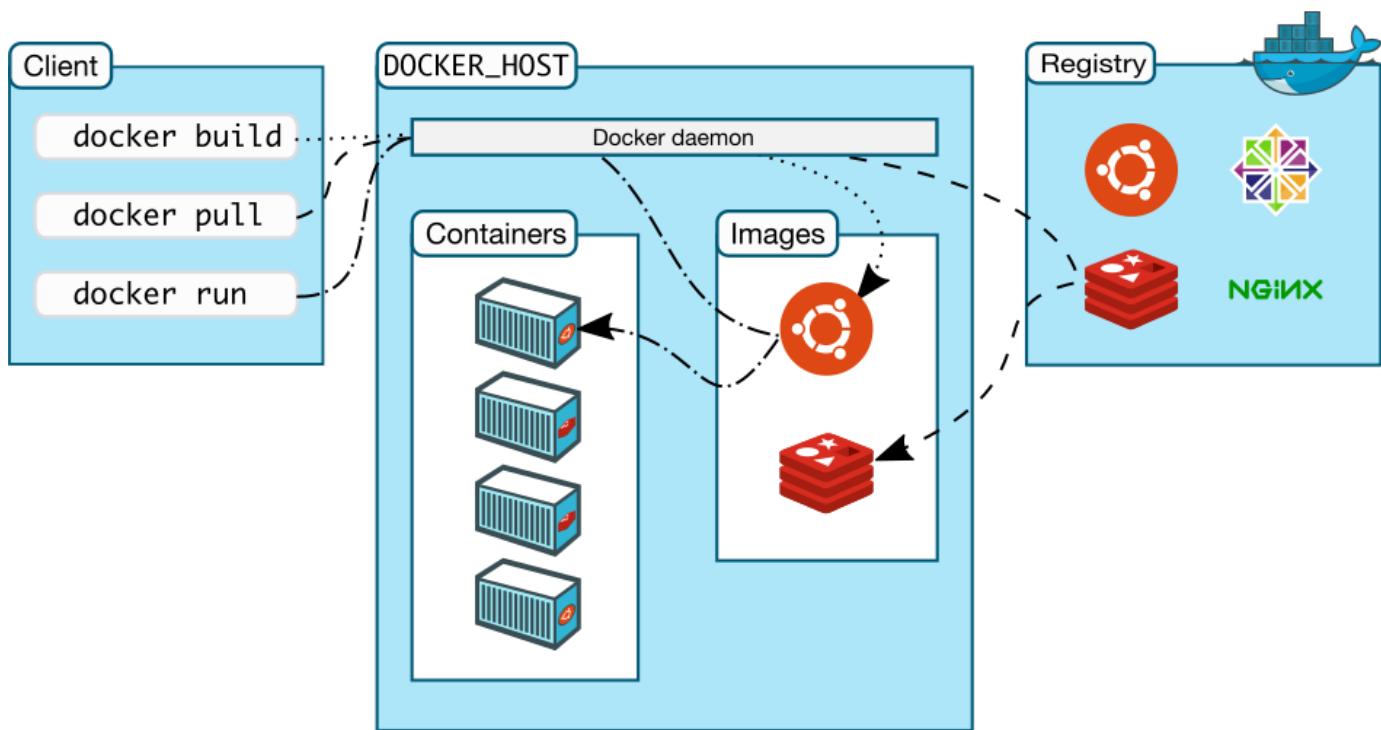
Платформа, основанная на контейнерах, позволяет легко портировать приложения. Контейнеры могут запускаться на локальной машине разработчика, в физических или виртуальных дата-центрах, облачных провайдерах или смешанных средах.

Запуск большого количества приложений на одной машине

Docker является легковесным и быстрым. Он предоставляет работоспособную и экономичную альтернативу виртуальным машинам на основе **гипервизора**, что позволяет использовать больше вычислительных мощностей для решения аналогичных задач.

Архитектура Docker

Docker использует клиент-серверную архитектуру. Клиент (**Docker client**) обращается к демону (**Docker daemon**), который поднимает (собирает), запускает и распределяет контейнеры. Клиент и демон могут быть запущены в одной системе или клиент может быть подключен к удаленному демону. Клиент и демон общаются через **REST API** поверх **UNIX-сокетов** или сетевого интерфейса. Другим клиентом является **Docker Compose**, позволяющий работать с приложениями, состоящими из нескольких контейнеров.



Демон

Демон (`dockerd`) регистрирует (слушает) запросы, поступающие от **Docker API**, и управляет такими объектами как образы, контейнеры, сети и тома. Демон может общаться с другими демонами для управления сервисами.

Клиент

Клиент (`docker`) - основной способ коммуникации с **Docker**. При выполнении такой команды, как `docker run`, клиент отправляет эту команду демону, который, собственно, эту команду и выполняет. Команда `docker` использует **Docker API**. Клиент может общаться с несколькими демонами.

Docker Desktop

Docker Desktop - это десктопное приложение для Mac, Windows и Linux, позволяющее создавать и распределять контейнерные приложения и микросервисы. Docker Desktop включает в себя демона, клиента, Docker Compose, Docker Content Trust, Kubernetes и Credential Helper.

Реестр

В реестре (registry) хранятся образы контейнеров. Docker Hub - это публичный реестр, который (по умолчанию) используется Docker для получения образов. Имеется возможность создания частных (закрытых) реестров.

При выполнении таких команд, как docker pull или docker run, необходимые образы загружаются из настроенного реестра. А при выполнении команды docker push образ загружается в реестр.

Объекты

При использовании Docker мы создаем и используем образы, контейнеры, сети, тома, плагины и другие объекты. Рассмотрим некоторые из них.

Образы (*Images*)

Образ - это доступный только для чтения шаблон с инструкциями по созданию контейнера. Часто образ представляет собой модификацию другого образа.

Можно создавать свои образы или использовать образы, созданные другими и опубликованные в реестре. Для создания образа используется Dockerfile, содержащий инструкции по созданию образа и его запуску (см. ниже). Ряд инструкций в Dockerfile приводит к созданию в образе нового слоя (раньше новый слой создавался для каждой инструкции). При изменении Dockerfile и повторной сборке образа пересобираются только модифицированные слои. Это делает образы легковесными, маленькими и быстрыми.

Контейнеры (*Containers*)

Контейнер - это запускаемый экземпляр образа. Мы создаем, запускаем, перемещаем и удаляем контейнеры с помощью Docker API или CLI (command line interface, интерфейс командной строки). Мы можем подключать контейнеры к сетям, добавлять в них хранилища данных и даже создавать новые образы на основе текущего состояния.

По умолчанию контейнеры хорошо изолированы от других контейнеров и хоста. Однако мы можем управлять тем, насколько изолированы сеть, хранилище данных или другая подсистема контейнера.

Контейнер определяется образом и настройками, указанными при его создании и запуске. При удалении контейнера его состояние также удаляется. Этого можно избежать с помощью хранилища данных.

Пример команды `docker run`

Следующая команда запускает контейнер `ubuntu`, интерактивно подключается к локальному сеансу командной строки и выполняет в ней команду `/bin/bash`:

```
docker run -i -t ubuntu /bin/bash
```

При выполнении этой команды происходит следующее:

1. Поскольку на нашей машине нет образа `ubuntu`, `Docker` загружает его из реестра (то же самое делает команда `docker pull ubuntu`).
2. `Docker` создает новый контейнер (то же самое делает команда `docker container create`).
3. В качестве последнего слоя `Docker` выделяет контейнеру файловую систему для чтения и записи. Это позволяет запущенному контейнеру создавать и модифицировать файлы и директории в его локальной файловой системе.
4. Поскольку мы не указали сетевых настроек, `Docker` создает сетевой интерфейс для подключения контейнера к дефолтной сети. Это включает в себя присвоение контейнеру `IP-адреса`. Контейнеры могут подключаться к внешним сетям через сетевое соединение хоста.
5. `Docker` запускает контейнер и выполняет `/bin/bash`. Поскольку контейнер запущен в интерактивном режиме и подключен к терминалу (благодаря флагам `-i` и `-t`), мы можем вводить команды и получать результаты в терминале.
6. Выполнение команды `exit` приводит к прекращению выполнения `/bin/bash`. Контейнер останавливается, но не удаляется. Мы можем запустить его снова или удалить.

Команды и флаги

`docker run`

Команда `docker run` используется для запуска контейнера. Это основная и потому наиболее часто используемая команда.

```
# сигнатура
docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
# основные настройки (флаги)
-d - запуск контейнера в качестве отдельного процесса
-p - публикация открытого порта в интерфейсе хоста (HOST:CONTAINER)
# например
-p 3000:3000
-t - выделение псевдотерминала
-i - оставить STDIN открытым без присоединения к терминалу
--name - название контейнера
--rm - очистка системы при остановке/удалении контейнера
--restart - политика перезапуска - по (default) | on-failure[:max-retries] |
always | unless-stopped
-e - установка переменной среды окружения
-v - привязка распределенной файловой системы (name:/path/to/file)
# например
-v mydb:/etc/mydb
-w - установка рабочей директории
```

Следующая команда запускает контейнер `postgres`:

```
# \ используется для разделения длинных команд на несколько строк
docker run --rm \
# название контейнера
--name postgres \
# пользователь
-e POSTGRES_USER=postgres \
# пароль
-e POSTGRES_PASSWORD=postgres \
# название базы данных
-e POSTGRES_DB=mydb \
# автономный режим и порт
-dp 5432:5432 \
# том для хранения данных
-v $HOME/docker/volumes/postgres:/var/lib/postgresql/data \
# образ
postgres
```

`docker build`

Команда `docker build` используется для создания образа на основе файла `Dockerfile` и контекста. Контекст - это набор файлов, находящихся в локации, определенной с помощью `PATH` или `URL`. `PATH` - это директория в нашей локальной системе, а `URL` - это удаленный репозиторий. Контекст сборки обрабатывается рекурсивно, поэтому `PATH` включает как директорию, там и все ее поддиректории, а `URL` - как репозиторий, так и все его субмодули. Для исключения файлов из сборки образа используется `.dockerignore` (синтаксис этого файла похож на `.gitignore`).

```
# сигнатура
docker build [OPTIONS] PATH | URL | -
```

Создание образа:

```
# в качестве контекста сборки используется текущая директория
docker build .
```

Использование репозитория в качестве контекста (предполагается, что `Dockerfile` находится в корневой директории репозитория):

```
docker build github.com/cr3ack/docker-firefox
```

```
docker build -f ctx/Dockerfile http://server/ctx.tar.gz
```

В данном случае `http://server/ctx.tar.gz` отправляется демону, которые загружает и извлекает файлы. Параметр `-f ctx/Dockerfile` определяет путь к `Dockerfile` внутри `ctx.tar.gz`.

Чтение `Dockerfile` из `STDIN` без контекста:

```
docker build - < Dockerfile
```

Добавление тега к образу:

```
docker build -t myname/my-image:latest .
```

Определение `Dockerfile`:

```
docker build -f Dockerfile.debug .
```

Экспорт файлов сборки в директорию `out`:

```
docker build -o out .
```

Экспорт файлов сборки в файл `out.tar`:

```
docker build -o - . > out.tar
```

`docker exec`

Команда `docker exec` используется для выполнения команды в запущенном контейнере.

```
# сигнатура
docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
# основные флаги
-d - выполнение команды в фоновом режиме
-e - установка переменной среды окружения
-i - оставить 'STDIN' открытым
-t - выделение псевдотерминала
-w - определение рабочей директории внутри контейнера
```

Пример:

```
# -U - это пользователь, которым по умолчанию является root
docker exec -it postgres psql -U postgres
```

В данном случае в контейнере `postgres` будет запущен интерактивный терминал `psql`.

Выполним парочку команд:

```
# получаем список баз данных
\l
# подключаемся к базе данных mydb
-d mydb
# получаем список таблиц, точнее, сообщение об отсутствии отношений (relations)
```

```
\dt  
# выходим  
\q
```

docker ps

Команда `docker ps` используется для получения списка (по умолчанию только запущенных) контейнеров.

```
# сигнатура  
docker ps [OPTIONS]  
# основные флаги  
-a - показать все контейнеры (как запущенные, так и остановленные)  
-f - фильтрация вывода на основе условия ('id', 'name', 'status' и т.д.)  
-l - показать п последних созданных контейнеров  
-l - показать последний созданный контейнер  
# пример получения списка приостановленных контейнеров  
docker ps -f 'status=paused'
```

Для получения списка образов используется команда `docker images`.

Команды управления

```
# запуск остановленного контейнера  
docker start CONTAINER  
  
# приостановление всех процессов, запущенных в контейнере  
docker pause CONTAINER  
  
# остановка контейнера  
docker stop CONTAINER  
  
# "убийство" контейнера  
docker kill CONTAINER  
  
# перезапуск контейнера  
docker restart CONTAINER  
  
# удаление остановленного контейнера  
docker rm [OPTIONS] CONTAINER  
# основные флаги  
-f - принудительное удаление (остановка и удаление) запущенного контейнера
```

```
-v - удаление анонимных томов, связанных с контейнером  
# пример удаления всех остановленных контейнеров  
docker rm $(docker ps --filter status=exited -q)  
  
# удаление образа  
docker rmi IMAGE  
  
###  
  
# управление образами  
docker image COMMAND  
  
# управление контейнерами  
docker container COMMAND  
  
# управление томами  
docker volume COMMAND  
  
# управление сетями  
docker network COMMAND  
  
# управление docker  
docker system COMMAND
```

Другие команды

Для получения логов запущенного контейнера используется команда `docker logs`:

```
docker logs [OPTIONS] CONTAINER  
# основные флаги  
-f - следование за выводом  
-n - n последних строк
```

Для удаления всех неиспользуемых данных (контейнеры, сети, образы и, опционально, тома) используется команда `docker system prune`. Основные флаги:

```
-a - удаление всех неиспользуемых образов, а не только обособленных (dangling)  
--volumes - удаление томов
```

Предостережение: применять эту команду следует с крайней осторожностью, поскольку удаленные данные не подлежат восстановлению.

Полный список команд и флагов.

Dockerfile

`Dockerfile` - это документ (без расширения), содержащий инструкции, которые используются для создания образа при выполнении команды `docker build`.

Предостережение: не используйте `/` в качестве `PATH` для контекста сборки. Это приведет к передаче демону всего содержимого жесткого диска вашей машины.

Инструкции выполняются по одной. Результаты наиболее важных инструкций фиксируются в виде отдельных слоев образа. *Обратите внимание:* каждая инструкция выполняется независимо от других. Это означает, что выполнение `RUN cd /tmp` не будет иметь никакого эффекта для последующих инструкций.

`Dockerfile` может содержать следующие инструкции:

```
# Кomentарий
ИНСТРУКЦИЯ аргументы

# Основные
# FROM - родительский образ
FROM <image>[:<tag>] [AS <name>]
# пример
FROM node:12-alpine AS build

# WORKDIR - установка рабочей директории для инструкций RUN, CMD, ENTRYPOINT,
COPY и ADD
WORKDIR /path/to/dir
# пример
WORKDIR /app

# COPY - копирование новых файлов или директорий из <src>
# и их добавление в файловую систему образа по адресу, указанному в <dest>
COPY <src> <dest>
COPY [<src>, <dest>]
# пример
COPY package.* уагп.lock ./
# или
COPY . .
```

```
# ADD, в отличие от COPY, позволяет копировать удаленные файлы,
# а также автоматически распаковывает сжатые (identity, gzip, bzip2 или xz)
локальные файлы

# ADD - копирование новых файлов, директорий или удаленного (!) файла из <src>
# и их добавление в файловую систему образа по адресу, указанному в <dest>
ADD <src> <dest>
ADD ["<src>", "<dest>"]
# пример
ADD some.txt some_dir/ # <WORKING_DIR>/some_dir/

# RUN - выполнение команды в новом слое на основе текущего образа и фиксация
результатата
RUN <command>
# или
RUN ["executable", "arg1", "arg2"] # Кавычки должны быть двойными
# пример
RUN npm install

# CMD - предоставление дефолтных значений исполняемому контейнеру
CMD ["executable", "arg1", "arg2"]
# или если данной инструкции предшествует инструкция ENTRYPOINT
CMD ["arg1", "arg2"]
# или
CMD command arg1 arg2
# пример
CMD [ "node", "/app/src/index.js" ]
# RUN выполняет команду и фиксирует результат,
# CMD ничего не выполняет во время сборки, а определяет команду для образа
# (!) выполняется только одна (последняя) инструкция CMD

# ENTRYPOINT - настройка исполняемого контейнера
ENTRYPOINT ["executable", "arg1", "arg2"]
ENTRYPOINT command arg1 arg2
# пример
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
# docker run -it --rm --name test top -H
# top -b -H
# разница между ENTRYPOINT и CMD:
# https://docs.docker.com/engine/reference/builder/#understand-how-cmd-and-
```

```
entrypoint-interact
# https://stackoverflow.com/questions/21553353/what-is-the-difference-between-
cmd-and-entrypoint-in-a-dockerfile

# переменные
# ${var} или $var
# пример
FROM busybox
ENV FOO=/bar
WORKDIR ${FOO}      # WORKDIR /bar
ADD . $FOO          # ADD . /bar
COPY \${FOO} /qux    # COPY ${FOO} /qux

# Другие
# LABEL - добавление метаданных к образу
LABEL <key>=<value>
# пример
LABEL version="1.0"

# EXPOSE - информация о сетевом порте, прослушиваемом контейнером во время
выполнения
EXPOSE <port> | <port>/<protocol>
# пример
EXPOSE 3000

# ENV - установка переменных среды окружения
ENV <key>=<value>
# пример
ENV MY_NAME="No Name"

# VOLUME - создание точки монтирования
VOLUME ["/var/log"]
VOLUME /var/log

# USER - установка пользователя для использования при запуске контейнера
# в любых инструкциях RUN, CMD и ENTRYPOINT
USER <user>[:<group>]
USER <UID>[:<GID>]
```

```
# ARG - определение переменной, которая может быть передана через командную строку при
# выполнении команды `docker build` с помощью флага `--build-arg <name>=<value>`
ARG <name>[=<default value>]

# ONBUILD - добавление в образ триггера, запускаемого при использовании
# данного образа в качестве основы для другой сборки
ONBUILD <INSTRUCTION>
```

[Справка по Dockerfile](#).

Рекомендации по Dockerfile

Рассмотрим следующий `Dockerfile`:

```
# syntax=docker/dockerfile:1
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Выполнение каждой инструкции (кроме `CMD`) из этого файла приводит к созданию нового слоя:

- `FROM` создает слой из образа `ubuntu:18.04`
- `COPY` добавляет файлы из текущей директории
- `RUN` собирает приложение с помощью `make`
- `CMD` определяет команду для запуска приложения в контейнере

При запуске образа и генерации контейнера мы добавляем новый слой, доступный для записи, поверх остальных. Все изменения в запущенном контейнере, такие как создание новых файлов, их модификация или удаление записываются в этот слой.

Создание эфемерных контейнеров

Генерируемые контейнеры должны быть максимально эфемерными. Под эфемерностью понимается возможность остановки, уничтожения, повторной сборки и замены контейнеров без необходимости дополнительной настройки процесса их генерации.

Понимание контекста сборки

При выполнении команды `docker build` контекстом сборки, как правило, является текущая директория. Предполагается, что `Dockerfile` находится в этой директории. Путь к `Dockerfile`, находящемуся в другом месте, можно указать с помощью флага `-f`. Независимо от того, где находится `Dockerfile`, все файлы и директории из текущей директории отправляются демону в качестве контекста сборки.

В следующем примере мы

- создаем (`mkdir`) директорию `myapp`, которая используется в качестве контекста сборки
- переходим в нее (`cd`)
- создаем файл `hello` с текстом "hello"
- создаем `Dockerfile`, читающий (`cat`) содержимое файла `hello`
- собираем образ с тегом `helloapp:v1`

```
mkdir myapp && cd myapp
echo "hello" > hello
echo -e "FROM busybox\nCOPY /hello /\nRUN cat /hello" > Dockerfile
docker build -t helloapp:v1 .
```

Размещаем `Dockerfile` и `hello` в разных директориях и собираем вторую версию образа без использования кеша предыдущей сборки (`-f` определяет путь к `Dockerfile`):

```
# создаем директории
mkdir -p dockerfiles context
# перемещаем файлы
mv Dockerfile dockerfiles && mv hello context
# собираем образ
docker build --no-cache -t helloapp:v2 -f dockerfiles/Dockerfile context
```

`.dockerignore`

В файле `.dockerignore` указываются файлы, не имеющие отношения к сборке и поэтому не включаемые в нее. Синтаксис `.dockerignore` похож на синтаксис `.gitignore` или `.npmignore`.

Многоэтапная сборка

Многоэтапная сборка позволяет существенно уменьшить размер финального образа без необходимости изучения процесса сборки на предмет наличия промежуточных слоев и файлов, которые можно удалить.

Если процесс сборки состоит из нескольких слоев, мы можем упорядочить их от редко модифицируемых до часто модифицируемых:

- установка инструментов, необходимых для сборки приложения
- установка или обновление зависимостей
- генерация приложения

Пример `Dockerfile` для Go-приложения:

```
# syntax=docker/dockerfile:1
FROM golang:1.16-alpine AS build

# устанавливаем инструменты
# выполняем `docker build --no-cache .` для обновления зависимостей
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

# список зависимостей из `Gopkg.toml` и `Gopkg.lock`
# эти слои будут собираться повторно только при изменении файлов `Gopkg`
COPY Gopkg.lock Gopkg.toml /go/src/project/
WORKDIR /go/src/project/
# устанавливаем зависимости
RUN dep ensure -vendor-only

# копируем проект и собираем его
# этот слой будет собираться повторно только при изменении файлов из директории
# `project`
COPY . /go/src/project/
RUN go build -o /bin/project

# получаем образ, состоящий из одного слоя
FROM scratch
COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]
```

Лишние библиотеки

Для уменьшения сложности, количества зависимостей и времени сборки следует избегать установки дополнительных и ненужных библиотек "на всякий случай".

Разделение приложений

Каждый контейнер должен иметь одну ответственность (*single responsibility*).

Разделение приложений на несколько контейнеров облегчает горизонтальное масштабирование и переиспользуемость контейнеров. Например, стек веб-приложения может состоять из 3 отдельных контейнеров, каждый со своим уникальным образом, для управления приложением, базы данных и сервера или распределенного кеша, хранящегося в памяти. Если контейнеры зависят друг от друга для обеспечения возможности их коммуникации следует использовать сети.

Минимизация количества слоев

В старых версиях `Docker` каждая инструкция в `Dockerfile` приводила к созданию нового слоя. Сейчас новые слои создаются только инструкциями `RUN`, `COPY` и `ADD`. Другие инструкции создают временные промежуточные образы, которые не приводят к увеличению размера сборки.

Сортировка многострочных аргументов

Многострочные аргументы рекомендуется сортировать в алфавитном порядке. Также рекомендуется добавлять пробел перед обратным слэшем (\). Пример:

```
RUN apt-get update && apt-get install -y \
bzr \
cvs \
git \
mercurial \
subversion \
&& rm -rf /var/lib/apt/lists/*
```

Использование кеша сборки

При сборке образа `Docker` изучает все инструкции в порядке, определенном в `Dockerfile`. После изучения инструкции `Docker` обращается к своему кешу. Если в кеше имеется соответствующий образ, новый образ не создается. Для сборки образа без обращения к кешу используется настройка `--no-cache=true`.

Рекомендации по инструкциям

FROM

В качестве основы для создания образа рекомендуется использовать официальные образы из [DockerHub](#) версии `alpine`.

LABEL

Подписи позволяют структурировать образы проекта, добавлять информацию о лицензиях, могут использоваться для автоматизации и т.д.

```
# одна подпись
LABEL com.example.version="0.0.1-beta"
# несколько подписей
LABEL vendor=ACME\ Incorporated \
      com.example.is-beta= \
      com.example.is-production="" \
      com.example.version="0.0.1-beta" \
      com.example.release-date="2021-01-12"
```

RUN

Длинные и сложные инструкции `RUN` рекомендуется разделять на несколько строк с помощью обратного слэша (`\`). Это делает `Dockerfile` более читаемым, облегчает его понимание и поддержку.

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo \
    && rm -rf /var/lib/apt/lists/*
```

CMD

Инструкция `CMD` используется для запуска программ в контейнере вместе с аргументами. `CMD` должна использоваться в форме `CMD ["executable", "param1", "param2"]`. В большинстве случаев первым элементом должен быть интерактивный терминал, такой как `bash`, `python` или `perl`. Например, `CMD ["perl", "-de0"]`, `CMD ["python"]` или `CMD ["php", "-a"]`. При использовании `ENTRYPOINT` следует убедиться, что пользователи понимают, как работает эта инструкция.

EXPOSE

Инструкция `EXPOSE` определяет порты, на которых контейнер регистрирует соединения. Рекомендуется использовать порты, которые являются традиционными для приложения. Например, образ, содержащий веб-сервер `Apache`, должен использовать `EXPOSE 80`, а образ, содержащий `MongoDB` - `EXPOSE 27017`.

ENV

Для облегчения запуска программы можно использовать `ENV` для обновления переменной среды окружения `PATH` для приложения, устанавливаемого контейнером. Например, `ENV PATH=/usr/local/nginx/bin:$PATH` обеспечивает, что `CMD ["nginx"]` просто работает.

Инструкция `ENV` также может быть полезна для предоставления обязательных для сервиса переменных, таких как `PGDATA` для `Postgres`.

Наконец, `ENV` может использоваться для установки номеров версий, что облегчает их обновление.

ADD или COPY

Хотя `ADD` и `COPY` имеют похожий функционал, в большинстве случаев следует использовать `COPY`, поскольку эта инструкция является более прозрачной, чем `ADD`. `COPY` поддерживает копирование в контейнер только локальных файлов, а `ADD` также позволяет извлекать файлы из локальных архивов и получать файлы по `URL`, но вместо последнего лучше использовать `curl` или `wget`: это позволяет удалять ненужные файлы после извлечения.

Если `Dockerfile` состоит из нескольких этапов, на которых используются разные файлы из контекста, эти файлы рекомендуется копировать индивидуально. Это позволяет обеспечить инвалидацию кеша только для модифицированных файлов. Например:

```
COPY package.json /app
RUN npm i
# предполагается, что директория node_modules указана в .dockerignore
COPY . /app
```

ENTRYPOINT

`ENTRYPOINT` определяет основную команду для образа, что позволяет запускать образ без этой команды.

Рассмотрим пример образа для инструмента командной строки `s3cmd`:

```
ENTRYPOINT ["s3cmd"]
CMD ["--help"]
```

Данный образ может быть запущен следующим образом:

```
docker run s3cmd
```

Это приведет к выводу справки.

Либо мы можем передать параметры для выполнения команды:

```
docker run s3cmd ls s3://mybucket
```

Это может быть полезным при совпадении названия образа со ссылкой на исполняемый файл.

VOLUME

Инструкция `VOLUME` следует использовать для доступа к любой области хранения базы данных, хранилищу настроек или файлам/директориям, созданным контейнером. Крайне не рекомендуется использовать `VOLUME` для мутабельных и/или пользовательских частей образа.

WORKDIR

Для ясности и согласованности для `WORKDIR` всегда следует использовать абсолютные пути. Также `WORKDIR` следует использовать вместо инструкций типа `RUN cd ... && do-something`, которые трудно читать, отлаживать и поддерживать.

Управление данными

По умолчанию файлы, создаваемые в контейнере сохраняются в слое, доступном для записи (`writable layer`). Это означает следующее:

- данные существуют только на протяжении жизненного цикла контейнера, и их сложно извлекать из контейнера, когда в них нуждается другой процесс

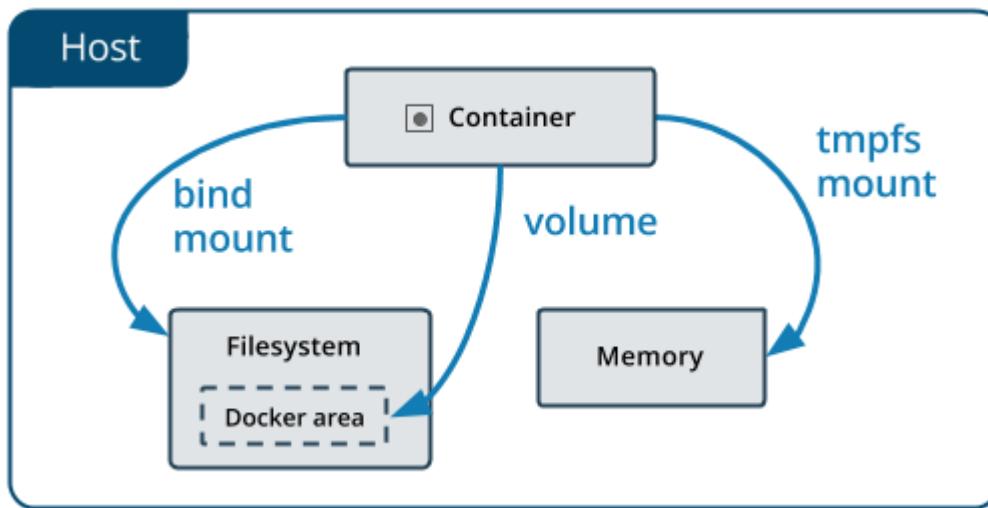
- слой контейнера, доступный для записи, тесно связан с хостом, на котором запущен контейнер. Данные нельзя просто взять и переместить в другое место
- запись в такой слой требует наличия драйвера хранилища (storage driver) для управления файловой системой. Эта дополнительная абстракция снижает производительность по сравнению с томами данных (data volumes), которые пишут напрямую в файловую систему

Docker предоставляет 2 возможности для постоянного хранения данных на хосте: тома (volumes) и **bind mount**. Для пользователей **Linux** также доступен **tmpfs mount**, а для пользователей **Windows** - **named pipe**.

Выбор правильного типа монтирования

Независимо от выбранного типа монтирования, для контейнера данные выглядят одинаково. Они представляют собой директорию или файл в файловой системе контейнера.

Разница между томами, **bind mount** и **tmpfs mount** заключается в том, где хранятся данные на хосте.



- тома хранятся в части файловой системы хоста, управляемой **Docker** (`/var/lib/docker/volumes/` на **Linux**). Процессы, не относящиеся к **Docker**, не должны модифицировать эту часть. Тома - лучший способ хранения данных в **Docker**
- **bind mount** может храниться в любом месте системы хоста. Это могут быть даже важные системные файлы и директории. Модифицировать их могут любые процессы
- **tmpfs mount** хранится в памяти и не записывается в файловую систему

Управление данными в **Docker**.

Тома

Тома (Volumes) - предпочтительный способ хранения данных, генерируемых и используемых контейнерами. Они полностью управляются Docker, в отличие от bind mount, которые зависят от структуры директории и операционной системы хоста. Преимущества томов состоят в следующем:

- тома легче восстанавливать и мигрировать
- томами можно управлять с помощью Docker CLI и Docker API
- тома работают как в Linux, так и в Windows контейнерах
- тома могут более безопасно распределяться между контейнерами
- движки томов (volume drivers) позволяют хранить тома на удаленных хостах и облачных провайдерах, шифровать содержимое томов или добавлять в них новый функционал
- содержимое новых томов может заполняться (population) контейнером
- тома имеют более высокую производительность

Тома также являются более предпочтительными перед хранением данных в слое контейнера, доступном для записи, поскольку тома не увеличивают размер контейнера. Содержимое контейнера существует за пределами жизненного цикла контейнера.

Флаги `-v` и `--mount`

`--mount` является более явным и многословным. Основное отличие состоит в том, что при использовании `-v` все настройки комбинируются вместе, а при использовании `--mount` они указываются раздельно.

Движок тома может быть определен только с помощью `--mount`.

- `-v` или `--volume`: состоит из 3 полей, разделенных двоеточием (`:`). Поля должны указываться в правильном порядке. Значение каждого поля не является очевидным
- в случае именованных томов первое поле - это название тома, которое является уникальным в пределах хоста. В случае анонимных томов это поле опускается
- второе поле - это путь файла или директории, монтируемой в контейнере
- третье поле является опциональным и представляет собой разделенный запятыми список настроек, таких как `ro`
- `--mount`: состоит из нескольких пар ключ/значение, разделенных запятыми.

Синтаксис `--mount` является более многословным, зато порядок ключей не имеет значения, а значения ключей являются более очевидными

- `type`: тип монтирования, может иметь значение `bind`, `volume` или `tmpfs`

- `source`: источник монтирования. Для именованных томов - это название тома. Для анонимных может быть опущено. Может определяться как `source` или `src`
- `destination`: путь монтируемого в контейнере файла или директории. Может определяться как `destination`, `dst` или `target`
- `readonly`: если указана данная настройка, том будет доступен только для чтения. Может определяться как `readonly` или `ro`
- `volume-opt`: данная настройка может определяться несколько раз, принимая пары название настройки/ее значение

`-v` и `--mount` принимают одинаковые настройки. При использовании томов с сервисами, доступен только флаг `--mount`.

Создание и управление томами

В отличие от `bind mount`, тома могут создаваться и управляться за пределами любого контейнера.

Создание тома

```
docker volume create my-vol
```

Список томов

```
docket volume ls
```

Анализ тома

```
docker volume inspect my-vol
```

Удаление тома

```
docker volume rm my-vol
```

Запуск контейнера с томом

При запуске контейнера с несуществующим томом, он создается автоматически. В следующем примере том `myvol2` монтируется в директорию `app` контейнера:

```
docker run -d \
--name devtest \
-v myvol2:/app \
nginx:latest

# или

docker run -d \
--name devtest \
--mount source=myvol2,target=/app \
nginx:latest
# далее будет использоваться только '-v'
```

В данном случае том будет доступен как для чтения, так и для записи.

Использование тома с Docker Compose

Единичный сервис (single service) **Compose** с томом может выглядеть так:

```
version: "1.0"
services:
  frontend:
    image: node:lts
    volumes:
      - myapp:/home/node/app
volumes:
  myapp:
```

Том будет создан при первом вызове **docker compose up**. При последующих вызовах данный том будет использоваться повторно.

Том может быть создан отдельно с помощью **docker volume create**. В этом случае в **docker-compose.yml** может быть указана ссылка на внешний (external) том:

```
version: "1.0"
services:
  frontend:
    image: node:lts
    volumes:
      - myapp:/home/node/app
volumes:
```

```
myapp:
  external: true
```

Популяция тома с помощью контейнера

При запуске нового контейнера, создающего том, когда контейнер содержит файлы и директории в монтируемой директории, содержимое этой директории копируется в том. После этого контейнер монтирует и использует том. Другие контейнеры, использующие том, будут иметь доступ к его предварительно заполненному (pre-populated) содержимому.

В следующем примере мы создаем контейнер `nginxtest` и заполняем новый том `nginx-vol` содержимым из директории `/usr/share/nginx/html`, в которой хранится дефолтная разметка:

```
docker run -d \
--name=nginxtest \
-v nginx-vol:/usr/share/nginx/html \
nginx:latest
```

Пример мониторинга тома, доступного только для чтения

```
# :go
docker run -d \
--name=nginxtest \
-v nginx-vol:/usr/share/nginx/html:ro \
nginx:latest
```

Резервное копирование, восстановление и передача данных томов

Для создания контейнера, монтирующего определенный том, используется флаг `--volumes-from`.

Резервное копирование тома

Создаем контейнер `dbstore`:

```
docker run -v /dbdata --name dbstore ubuntu /bin/bash
```

Следующая команда

- запускает новый контейнер и монтирует в него том из контейнера `dbstore`
- монтирует директорию из локального хоста как `/backup`
- передает команду для архивации содержимого тома `dbdata` в файл `backup.tar` в директории `/backup`:

```
docker run --rm \
--volumes-from dbstore \
-v $(pwd):/backup \
ubuntu tar \
cvf /backup/backup.tar /dbdata
```

Восстановление данных тома

Создаем новый контейнер:

```
docker run -d \
-v /dbdata \
--name dbstore2 \
ubuntu /bin/bash
```

Распаковываем архив в том нового контейнера:

```
docker run --rm \
--volumes-from dbstore2 \
-v $(pwd):/backup \
ubuntu bash -c \
"cd /dbdata && tar xvf /backup/backup.tar --strip 1"
```

Удаление томов

Данные томов сохраняются после удаления контейнеров. Существует 2 типа томов:

- именованные: имеют определенный источник за пределами контейнера, например, `awesome:/foo`
- анонимные: не имеют определенного источника, поэтому при удалении контейнера демону следует передавать инструкции по их удалению

Для удаления анонимного контейнера можно использовать флаг `--rm`. Например, здесь мы создаем анонимный том `/foo` и именованный том `awesome:/bar`:

```
docker run --rm -v /foo -v awesome:/bar busybox top
```

После удаления этого контейнера будет удален только том `/foo`.

Для удаления всех неиспользуемых томов используется команда `docker volume prune`.

Более подробную информацию о томах можно получить [здесь](#).

Сети

Сети (Networks) позволяют контейнерам общаться между собой. Сетевой интерфейс Docker не зависит от платформы.

Основная функциональность сетей представлена следующими драйверами:

- `bridge` (мост): дефолтный сетевой драйвер. Такие сети, как правило, используются, когда приложение состоит из нескольких автономных контейнеров, которые должны иметь возможность общаться между собой
- `host`: этот драйвер также предназначен для автономных контейнеров, он позволяет использовать сети хоста напрямую, удаляя изоляцию между контейнером и хостом
- `overlay` (перекрытие): такие сети объединяют нескольких демонов вместе и позволяют групповым сервисам (swarm services) взаимодействовать друг с другом. Эти сети также могут использоваться для обеспечения коммуникации между групповым сервисом и отдельным контейнером или между двумя автономными контейнерами на разных демонах
- `ipvlan`: такая сеть предоставляет пользователю полный контроль над адресацией IPv4 и IPv6
- `macvlan`: этот драйвер позволяет присваивать контейнеру MAC-адрес, в результате чего контейнер появляется в сети как физическое устройство
- `none`: отключает сети для данного контейнера. Обычно, используется в дополнение к кастомному сетевому драйверу
- плагины

Резюме

- `bridge`: для обеспечения взаимодействия контейнеров, находящихся на одном хосте

- `host`: когда сетевой стек не должен быть изолирован от хоста
- `overlay`: для обеспечения взаимодействия контейнеров, находящихся на разных хостах, или для приложений, работающих вместе через групповые сервисы

Обзор сетей.

Использование дефолтной сети `bridge`

Сейчас мы запустим 2 контейнера `alpine` и посмотрим, как они могут взаимодействовать между собой.

- Открываем терминал и выполняем следующую команду:

```
docker network ls
```

Получаем список сетей `bridge`, `host` и `none`.

- Запускаем 2 контейнера `alpine`. `ash` - это дефолтный терминал `Alpine` (аналог `bash`). Флаги `-dit` означают запуск контейнера в фоновом режиме (`-d`), интерактивно (чтобы дает возможность вводить команды) (`-i`) и с псевдотерминалом (чтобы видеть ввод и вывод) (`-t`). Поскольку мы не указываем `--network`, контейнеры подключаются к дефолтной сети `bridge`:

```
# контейнер номер раз
docker run -dit --name alpine1 alpine ash
```

```
# и номер два
docker run -dit --name alpine2 alpine ash
```

Проверяем, что оба контейнера запущены:

```
docker container ls
# или
docker ps
```

- Анализируем сеть `bridge` на предмет подключенных к ней контейнеров:

```
docker network inspect bridge
```

Видим созданные нами контейнеры в разделе "Containers" вместе с их IP-адресами (172.17.0.2 для alpine1 и 172.17.0.3 для alpine2).

- Подключаемся к alpine1:

```
docker attach alpine1
```

Видим приглашение #, свидетельствующее о том, что мы являемся пользователем root внутри контейнера. Взглянем на сетевой интерфейс alpine1:

```
id addr show
```

Первый интерфейс (lo) нас не интересует. Нас интересует IP-адрес второго интерфейса (172.17.0.2).

- Проверяем подключение к Интернету:

```
ping -c 2 google.com
```

Флаг -c 2 означает 2 попытки ping.

- Обратимся ко второму контейнеру. Сначала по IP-адресу:

```
ping -c 2 172.17.0.3
```

Работает. Теперь попробуем обратиться по названию контейнера:

```
ping -c 2 alpine2
```

Не работает.

- Останавливаем и удаляем контейнеры:

```
docker container stop alpine1 alpine2  
docker container rm alpine1 alpine2
```

Обратите внимание: дефолтная сеть `bridge` не рекомендуется для использования в продакшне.

Использование кастомной сети `bridge`

В следующем примере мы снова создаем 2 контейнера `alpine`, но подключаем их к кастомной сети `alpine-net`. Эти контейнеры не подключены к дефолтной сети `bridge`. Затем мы запускаем третий `alpine`, подключаемый к `bridge`, но не к `alpine-net`, и четвертый `alpine`, подключаемый к обеим сетям.

- Создаем сеть `alpine-net`. Флаг `--driver bridge` можно опустить, поскольку `bridge` является сетью по умолчанию:

```
docker network create --driver bridge alpine-net
```

- Получаем список сетей:

```
docker network ls
```

Анализируем сеть `alpine-net`:

```
docker network inspect alpine-net
```

Получаем `IP-адрес` сети и пустой список подключенных контейнеров.

Обратите внимание на сетевой шлюз, который отличается от шлюза `bridge`.

- Создаем 4 контейнера. *Обратите внимание* на флаг `--network`. При выполнении команды `docker run` мы можем подключить контейнер только к одной сети, поэтому подключение `alpine4` к `alpine-net` выполняется отдельно:

```
docker run -dit --name alpine1 --network alpine-net alpine ash
```

```
docker run -dit --name alpine2 --network alpine-net alpine ash
```

```
docker run -dit --name alpine3 alpine ash
```

```
docker run -dit --name alpine4 alpine ash
```

```
# !  
docker network connect alpine-net alpine4
```

Получаем список запущенных контейнеров:

```
docker ps
```

- Анализируем `bridge` и `alpine-net`:

```
docker network inspect bridge
```

Видим, что к данной сети подключены контейнеры `alpine3` и `alpine4`.

```
docker network inspect alpine-net
```

А к этой сети подключены контейнеры `alpine1`, `alpine2` и `alpine4`.

- В кастомных сетях контейнеры могут обращаться друг к другу не только по IP-адресам, но также по названиям. Данная возможность называется автоматическим обнаружением сервиса (automatic service discovery). Подключаемся к `alpine1`:

```
docker container attach alpine1
```

```
ping -c 2 alpine2  
# success
```

```
ping -c 2 alpine 4  
# success
```

```
ping -c 2 alpine1  
# success
```

- Пробуем обратиться к `alpine3`:

```
ping -c 2 alpine3  
# failure
```

- Отключаемся от `alpine1` и подключаемся к `alpine4`:

```
docker container attach alpine4
```

```
ping -c 2 alpine1
# success
```

```
ping -c 2 alpine2
# success
```

```
ping -c 2 alpine3
# failure
# но
ping -c 2 172.17.0.2 # IP-адрес alpine3
# success
```

```
ping -c 2 alpine4
# success
```

- Все контейнеры могут подключаться к Интернету:

```
ping -c 2 google.com
# success
```

- Останавливаем и удаляем контейнеры и сеть `alpine-net`:

```
docker container stop alpine1 alpine2 alpine3 alpine4
```

```
docker container rm alpine1 alpine2 alpine3 alpine4
```

```
docker network rm alpine-net
```

Рассмотрим парочку примеров `Dockerfile` для `Node.js`-приложений.

Пример с официального сайта `Node.js`

```
FROM node:16
```

```
# создание директории приложения
WORKDIR /usr/src/app
```

```

# установка зависимостей
# символ астериск ("*") используется для того чтобы по возможности
# скопировать оба файла: `package.json` и `package-lock.json`
COPY package*.json ./

RUN npm install
# для создания сборки для продакшн
# RUN npm ci --only=production

# копируем исходный код
COPY . .

EXPOSE 4000
CMD [ "node", "server.js" ]

```

Инструкции `COPY package*.json ./` и `COPY . .` выполняются по отдельности в целях извлечения максимальной выгоды из кеширования слоев. Зависимости проекта меняются на так часто, как файлы, не имеет смысла устанавливать их при каждой сборке образа.

Пример из статьи "10 лучших практик по контейнеризации Node.js-приложений с помощью Docker"

Пример является сокращенным и для продакшна.

```

FROM node:lts-
alpine@sha256:b2da3316acdc2bec442190a1fe10dc094e7ba4121d029cb32075ff59bb27390a
RUN apk add dumb-init
ENV NODE_ENV production
WORKDIR /usr/src/app
COPY --chown=node:node . .
RUN npm ci --only=production
USER node
CMD ['dumb-init', 'node', 'server.js']

```

- Официальное руководство для [Node.js](#)
- Лучшие практики для [Node.js](#)

Docker Compose

Docker Compose - это инструмент для определения и запуска **Docker-приложений**, состоящих из нескольких контейнеров. Для настройки сервисов приложения используется файл `docker-compose.yml`.

Процесс использования **Compose**, как правило, состоит из 3 этапов:

- определение среды приложения с помощью **Dockerfile**;
- определение сервисов, из которых состоит приложение, в `docker-compose.yml` (для совместного запуска сервисов в изолированной среде);
- выполнение команды `docker compose up` для запуска приложения.

Пример файла `docker-compose.yml`:

```
version: "3.9" # опционально
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./code
      - logvolume:/var/log
    links:
      - redis
  redis:
    image: redis
    volumes:
      logvolume: {}
```

Compose позволяет делать следующее:

- запускать, останавливать и повторно собирать сервисы;
- получать статус запущенных сервисов;
- получать логи запущенных сервисов;
- выполнять команды в сервисах.

Compose предоставляет следующие возможности:

- создание нескольких изолированных сред на одном хосте;
- сохранение данных томов при создании контейнеров;

- повторное создание только модифицированных контейнеров;
- передача переменных среды окружения и возможность создания разных сред (для разработки, продакшна и т.д.).

Начало работы с Docker Compose.

docker compose

Команда `docker compose` является альтернативой `docker-compose CLI` и используется для управления `Compose`.

```
# сигнатура
docker-compose [-f <arg>...] [--profile <name>...] [options] [COMMAND] [ARGS...]

# основные флаги
-f - путь к docker-compose.yml
-p - название проекта
--project-path - альтернативная рабочая директория (по умолчанию рабочей является директория, содержащая docker-compose.yml)

# основные команды
up - создание и запуск сервисов
down - остановка и удаление контейнеров, сетей, образов и томов
start - запуск сервисов
stop - остановка сервисов
restart - перезапуск сервисов
create - создание сервисов
rm - удаление остановленных контейнеров
run - выполнение одноразовой команды
exec - выполнение команды в запущенном контейнере
```

Полный список флагов и команд.

docker-compose.yml

Файл `Compose` - это файл в формате `YAML`, определяющий сервисы, сети и тома.

Дефолтным путем этого файла является `./docker-compose.yml`.

Определение сервиса включает в себя установку настроек, которые применяются к каждому контейнеру, запущенному для этого сервиса. Это похоже на передачу аргументов при

выполнении команды `docker run`. Определения сети и тома аналогично выполнению команд `docker network create` и `docker volume create`.

Настройки, определенные в `Dockerfile`, такие как `CMD`, `EXPOSE`, `VOLUME` и `ENV` не нуждаются в дублировании в `docker-compose.yml`.

Рассмотрим основные настройки сервисов.

build

Настройки, применяемые во время сборки.

`build` может определяться в виде строки - пути к контексту сборки:

```
version: "3.9"
services:
  webapp:
    build: ./dir
```

Или в виде объекта, где `context` - путь к контексту, `dockerfile` - используемый `Dockerfile` и `args` - аргументы:

```
version: "3.9"
services:
  webapp:
    build:
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        buildno: 1
```

В случае с `args` аргументы должны быть определены в `Dockerfile`:

```
# syntax=docker/dockerfile:1

ARG buildno
ARG gitcommithash

RUN echo "Номер сборки: $buildno"
RUN echo "Основано на коммите: $gitcommithash"
```

```
build:
  context: .
  args:
    buildno: 1
    gitcommithash: cdc3b19
    # or
    - buildno=1
    - gitcommithash=cdc3b19
```

network

Сеть, к которой подключается контейнер во время сборки (для использования при выполнении команды `RUN`):

```
build:
  context: .
  network: host
# or
build:
  context: .
  network: custom_network_1
```

command

Перезапись дефолтной команды:

```
command: bundle exec thin -p 3000
```

depends_on

Зависимость между сервисами. Это означает следующее:

- `docker compose up` запускает сервисы в определенном порядке. В примере ниже `db` и `redis` запускаются перед `web`;
- `docker compose up SERVICE` автоматически включает зависимости `SERVICE`. В примере `docker compose up web` также создает и запускает `db` и `redis`;
- `docker compose stop` останавливает сервисы в определенном порядке. В примере `web` останавливается перед `db` и `redis`.

```

version: "3.9"
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres

```

restart_policy

Политика перезапуска - определяет, как и когда контейнер должен перезапускаться:

- `condition`: условие перезапуска - `none`, `on-failure` или `any` (значение по умолчанию);
- `delay`: время между попытками (по умолчанию равняется `5s`);
- `max_attempts`: количество попыток (по умолчанию - бесконечное);
- `window`: время принятия решения об успехе перезапуска (по умолчанию - немедленно).

```

version: "3.9"
services:
  redis:
    image: redis:alpine
    deploy:
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3
        window: 120s

```

entrypoint

Перезапись дефолтной точки входа:

```
entrypoint: /code/entrypoint.sh
```

env_file

Извлечение переменных среды окружения из файла. Может быть единичным значением или списком.

Если файл `Compose` определен с помощью `docker compose -f FILE`, пути в `env_file` будут относительными директориями, в которой находится этот файл.

Переменные, определенные в разделе `environment`, перезаписывают эти значения.

```
env_file: .env
# or
env_file:
- ./common.env
- ./apps/web.env
- /opt/runtime_opts.env
```

expose

Выставление портов без их публикации на хосте - порты будут доступны только связанным (linked) сервисам. Могут определяться только внутренние порты:

```
expose:
- "3000"
- "8000"
```

external_links

Подключение к контейнеру, запущенному за пределами `docker-compose.yml` или даже за пределами `Compose`. Особенno полезно для контейнеров, предоставляющих общие или распределенные сервисы:

```
external_links:
- redis_1
- project_db_1:mysql
- project_db_1:postgresql
```

Обратите внимание: внешние контейнеры должны быть подключены хотя бы к одной сети, к которой подключен сервис.

image

Образ для контейнера. Может быть репозиторием/тегом или частичным идентификатором (partial identifier):

```
image: redis
image: node:16
image: example-registry.com:4000/postgresql
```

links

Подключение контейнера к другому сервису. Подключаемый сервис определяется с помощью названия сервиса и синонима ссылки (link alias) ("SERVICE:ALIAS") или только названия:

```
web:
  links:
    - "db"
    - "db:database"
    - "redis"
```

network_mode

Сетевой режим:

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
```

networks

Сети для подключения:

```
services:
  some-service:
    networks:
      - some-network
      - other-network
```

ports

Выставление портов.

Короткий синтаксис позволяет делать следующее:

- определять оба порта (`HOST:CONTAINER`);
- определять только порт контейнера (для хоста выбирается эфемерный порт);
- определять `IP-адрес` хоста для привязки (`bind`) и оба порта (значением по умолчанию является `0.0.0.0`, что означает все интерфейсы) (`IPADDR:HOSTPORT:CONTAINERPORT`).

`ports:`

```
- "3000"
- "8000:8000"
- "9090-9091:8080-8081"
- "127.0.0.1:8001:8001"
- "127.0.0.1::5000"
- "6060:6060/udp"
```

Длинный синтаксис позволяет настраивать дополнительные поля:

- `target`: порт контейнера;
- `published`: порт хоста (доступный публично);
- `protocol`: протокол порта (`tcp` или `udp`);
- `mode`: `host` | `ingress`.

`restart`

Определение политики перезапуска. Значением по умолчанию является `no`, что означает отключение автоматического перезапуска. `always` означает перезапуск в любом случае. `on-failure` означает перезапуск только в случае аварийной остановки контейнера. `unless-stopped` означает перезапуск контейнера во всех случаях, кроме преднамеренной остановки:

```
restart: "no"
restart: always
restart: on-failure
restart: unless-stopped
```

volumes

Монтирование путей хоста (host paths) или именованных томов (named volumes), определенных в виде дополнительных настроек сервиса.

Пути хоста могут монтироваться как часть определения сервиса. Их не нужно указывать в ключе `volume` на верхнем уровне.

Однако, если необходимо, чтобы тома использовались несколькими сервисами, они должны быть перечислены в таком `volume`.

В следующем примере именованный том `mydata` используется сервисом `web`, для отдельного сервиса определяется `bind mount` (первый путь в `volumes` сервиса `db`). `db` также использует именованный том `dbdata` (второй путь), но определяет его с помощью устаревшего строкового формата для монтирования именованных томов.

Именованные тома указываются в ключе `volume` верхнего уровня:

```
version: "3.9"
services:
  web:
    image: nginx:alpine
    volumes:
      - type: volume
        source: mydata
        target: /data
        volume:
          nocopy: true
      - type: bind
        source: ./static
        target: /opt/app/static

  db:
    image: postgres:latest
    volumes:
      - "/var/run/postgres/postgres.sock:/var/run/postgres/postgres.sock"
      - "dbdata:/var/lib/postgresql/data"

volumes:
  mydata:
  dbdata:
```

Короткий синтаксис

В этом случае используется формат `[SOURCE:]TARGET[:MODE]`, где `SOURCE` - это путь хоста или именованный том, `TARGET` - это путь монтирования тома в контейнере и `MODE` - `ro` для доступа только для чтения и `rw` для доступа для чтения и записи (дeфолтное значение).

Для монтирования могут использоваться относительные пути (путь вычисляется, начиная с директории с файлом `Compose`). Относительные пути должны начинаться с `.` или `..`.

volumes:

определяем только путь и делегируем создание тома движку
- /var/lib/mysql

определяем связывание (mapping) абсолютных путей
- /opt/data:/var/lib/mysql

путь хоста относительно директории с файлом 'Compose'
- ./cache:/tmp/cache

путь относительно пользователя
- ~/configs:/etc/configs/:ro

именованный том
- datavolume:/var/lib/mysql

Длинный синтаксис

Длинный синтаксис позволяет настраивать дополнительные поля:

- `type`: тип монтирования - `volume`, `bind`, `tmpfs` или `none`;
- `source`: источник монтирования, путь хоста для `bind mount` или название тома, определенное в верхнеуровневом `volumes`;
- `target`: путь монтирования тома в контейнере;
- `read_only`: индикатор доступности тома только для чтения;
- `bind`: дополнительные настройки связывания:
- `propagation`: режим распространения, используемый для связывания;
- `volume`: дополнительные настройки тома:
- `no_copy`: индикатор запрета копирования данных тома.

```
version: "3.9"
services:
  web:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - type: volume
        source: mydata
        target: /data
      volume:
        nocopy: true
      - type: bind
        source: ./static
        target: /opt/app/static

networks:
  webnet:

volumes:
  mydata:
```

Другие настройки, соответствующие настройкам команды `docker run`

```
user: postgres
working_dir: /code

domainname: foo.com
hostname: foo
ipc: host
mac_address: 02:42:ac:11:65:43

privileged: true

read_only: true
shm_size: 64M
stdin_open: true
tty: true
```

Примеры определения продолжительности

2.5s
10s
1m30s
2h32m
5h34m56s

Примеры определения байтовых значений

2b
1024kb
2048k
300m
1gb

Замена переменных

Настройки могут содержать переменные среды окружения. Compose использует значения переменных из терминала при выполнении команды docker compose. Например, предположим, что терминал содержит POSTGRES_VERSION=9.3 и применяется такая настройка:

```
db:  
  image: "postgres:${POSTGRES_VERSION}"
```

При выполнении docker compose значение переменной POSTGRES_VERSION в настройках заменяется на 9.3 и мы получаем postgres:9.3.

Если значение переменной не установлено, переменная в настройках заменяется пустой строкой и мы получаем postgres:.

Дефолтные значения переменных могут быть установлены в файле .env, который должен находиться в той же директории, что и файл Compose. Значения переменных в терминале перезаписывают значения, определенные в .env.

Поддерживается 2 варианта синтаксиса: \$VAR и \${VAR}. Второй вариант предоставляет такие дополнительные возможности, как:

- определение значений по умолчанию:

- `${VAR:-default}`: оценивается как `default`, когда `VAR` не установлена или является пустой;
- `${VAR-default}`: оценивается как `default` только когда `VAR` не установлена;
- определение обязательных значений:
- `${VAR:?error}`: ошибка возникает, если `VAR` не установлена или является пустой;
- `${VAR?error}`: ошибка возникает, только если `VAR` не установлена.

Расширенные возможности интерполяции переменных типа `${VAR/foo/bar}` в настоящее время не поддерживаются.

Спецификация файла `Compose` версии 3.

Разработка приложения

Подготовка и настройка проекта

Предполагается, что вы хотя бы вкратце ознакомились с содержанием предыдущих частей или изучали другие материалы, посвященные работе с `Docker`. Впрочем, в этой части `Docker` будет совсем чуть-чуть.

Также предполагается, что на вашей машине установлен `Docker` и `Node.js`.

Хорошо, если на вашей машине установлен `Yarn` и вы имеете опыт работы с `React`, `Vue`, `Node.js`, `PostgreSQL` и `sh` или `bash` (все это опционально).

Как я сказал, наше приложение будет состоять из трех сервисов:

- клиента на `React.js`;
- админки на `Vue.js`;
- сервера (API) на `Node.js`.

В качестве базы данных мы будем использовать `PostgreSQL`, а для взаимодействия с ней - `Prisma`.

Функционал нашего приложения будет следующим:

- в админке задаются настройки для приветствия, темы и базового размера шрифта;
- эти настройки записываются в БД и применяются на клиенте;
- на клиенте реализована "тудушка";

- задачи записываются в БД;
- все это обслуживается сервером.

Создаем директорию для проекта, переходим в нее и создаем еще парочку директорий:

```
mkdir docker-test
cd !$ # docker-test
mkdir services sh uploads
```

В директории `services` будут находиться наши сервисы, в директории `sh` - скрипты для терминала, директорию `uploads` мы использовать не будем, но обычно в ней хранятся различные файлы, загружаемые админом или пользователями.

Переходим в директорию `services`, создаем директорию для API, генерируем шаблон клиента с помощью `Create React App` и шаблон админки с помощью `Vue CLI`:

```
cd services
mkdir api
yarn create react-app client
# or
npx create-react-app client

yarn create vue-app admin
# or
npx vue create admin
```

Начнем с API.

API

Переходим в директорию `api`, инициализируем `Node.js`-проект и устанавливаем зависимости:

```
cd api
yarn init -y
```

```
# от
npm init -y

# производственные зависимости
yarn add express cors
# зависимости для разработки
yarn add -D nodemon prisma
```

- `express` - `Node.js`-фреймворк для разработки веб-серверов;
- `cors` - утилита для работы с `CORS`;
- `nodemon` - утилита для запуска сервера для разработки;
- `prisma` - ядро (`core`) `ORM`, которое мы будем использовать для взаимодействия с `postgres`.

Инициализируем `prisma`:

```
npx prisma init
```

Это приводит к генерации директории `prisma`, а также файлов `prisma/schema.prisma` и `.env`.

Определяем генератор, источник данных и модели в файле `prisma/schema.prisma`:

```
// https://pris.ly/d/prisma-schema
generator client {
  provider      = "prisma-client-js"
  // это нужно для контейнера
  binaryTargets = ["native"]
}

datasource db {
  provider = "postgresql"
  // путь к БД извлекается из переменной среды окружения `DATABASE_URL`
  url      = env("DATABASE_URL")
}

// модель для настроек
model Settings {
  id          Int    @id @default(autoincrement())
  created_at  DateTime @default(now())
  updated_at  DateTime @updatedAt
```

```

greetings      String
theme          String
base_font_size String
}

// модель для задачи
model Todo {
    id          Int      @id @default(autoincrement())
    created_at DateTime @default(now())
    updated_at DateTime @updatedAt
    text        String
    done        Boolean
}

```

Определяем путь к БД в файле `.env`:

```
DATABASE_URL=postgresql://postgres:postgres@localhost:5432/mydb?schema=public
```

Здесь:

- `postgres` - имя пользователя и пароль;
- `localhost` - хост, на котором запущен сервер `postgres`;
- `5432` - порт, на котором запущен сервер `postgres`;
- `mydb` - название БД.

Определяем команду для запуска контейнера `postgres` в файле `sh/db` (без расширения):

```
docker run --rm --name postgres -e POSTGRES_PASSWORD=postgres -e
POSTGRES_USER=postgres -e POSTGRES_DB=mydb -dp 5432:5432 -v
$HOME/docker/volumes/postgres:/var/lib/postgresql/data postgres
```

Обратите внимание: если вы работаете на `Mac`, вам потребуется предоставить самому себе разрешение на выполнение кода из файла `sh/db`. Это можно сделать так:

```
# мы находимся в директории `sh`
chmod +x db
# or
sudo chmod +x db
```

Находясь в корневой директории проекта, открываем терминал и выполняем команду:

```
sh/db
```

Происходит загрузка образа `postgres` из [Docker Hub](#) и запуск контейнера под названием `postgres`.

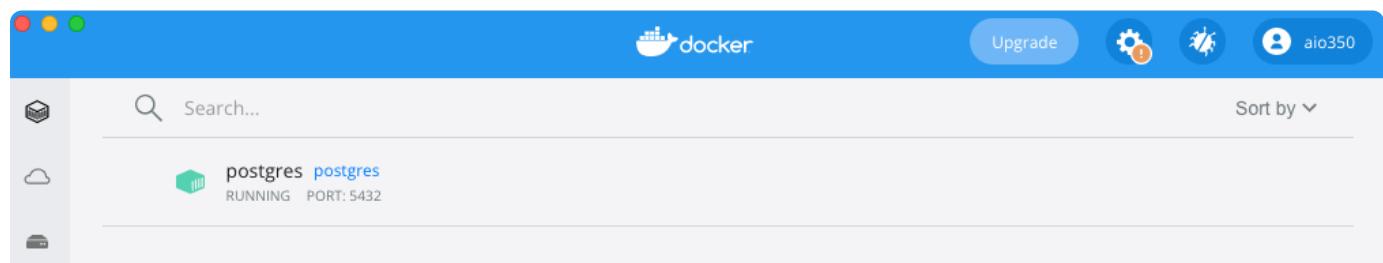
Обратите внимание: иногда может возникнуть ошибка, связанная с тем, что порт 5432 занят другим процессом. В этом случае необходимо найти `PID` данного процесса и "убить" его. На `Mac` это делается так:

```
# получаем 'PID' процесса, запущенного на порту '5432'
sudo lsof -i :5432
# предположим, что 'PID' имеет значение '103'
# "убиваем" процесс
sudo kill 103
```

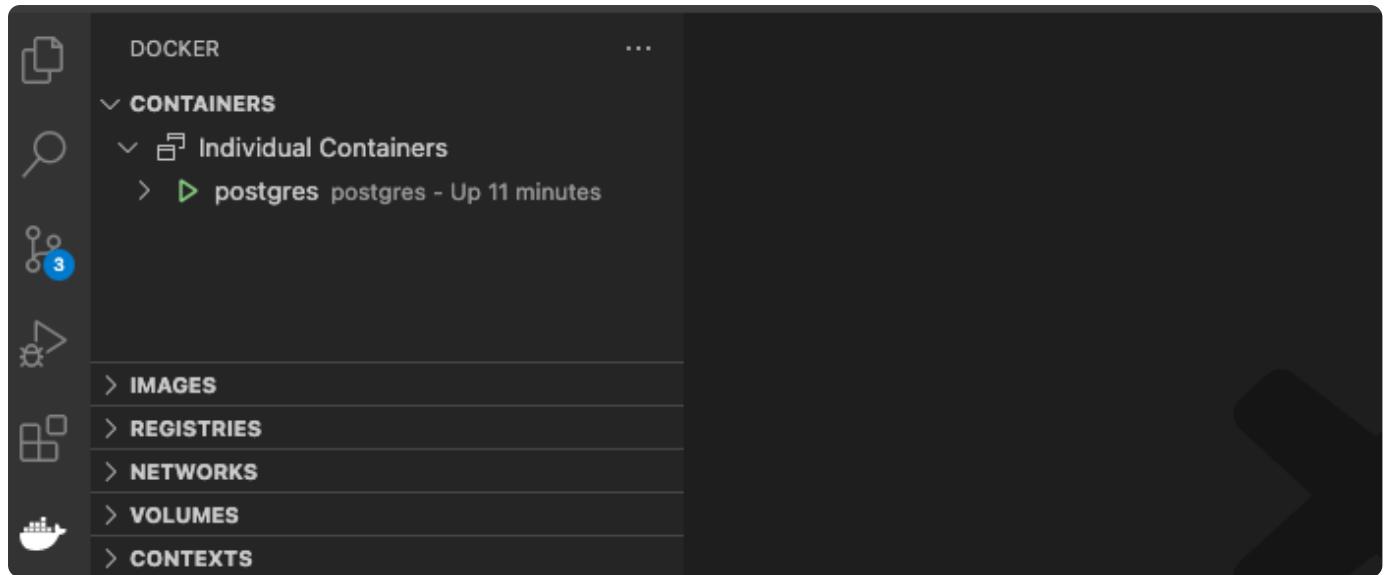
Убедиться в запуске контейнера можно, выполнив команду `docker ps`:

```
jetuser@JET100067-8 ~ % docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
0649a8829e40 postgres "docker-entrypoint.s..." About a minute ago Up About a minute 0.0.0.0:5432->5432/tcp postgres
```

Или запустив [Docker Desktop](#):



Или в разделе `Individual Containers` расширения [Docker](#) для [VSCode](#):



Выполняем миграцию:

```
# мы находимся в директории `api`
# migrate dev - миграция для разработки
# --name init - название миграции
npx prisma migrate dev --name init
```

Это приводит к генерации файла `prisma/migrations/[Date]-init/migration.sql`, подключению к БД, созданию в ней таблиц, установке и настройке `@prisma/client`.

Создаем файл `prisma/seed.js` с кодом для заполнения БД начальными данными:

```
import Prisma from '@prisma/client'
const { PrismaClient } = Prisma

// инициализируем клиента
const prisma = new PrismaClient()

// начальные настройки
const initialSettings = {
  greetings: 'Welcome to Docker Test App',
  theme: 'light',
  base_font_size: '16px'
}

// начальные задачи
const initialTodos = [
  {
    text: 'Eat',
    done: true
  }
]
```

```

},
{
  text: 'Code',
  done: true
},
{
  text: 'Sleep',
  done: false
},
{
  text: 'Repeat',
  done: false
}
]

async function main() {
  try {
    // если таблица настроек является пустой
    if (!(await prisma.settings.findFirst())) {
      await prisma.settings.create({ data: initialSettings })
    }

    // если таблица задач является пустой
    if (!(await (await prisma.todo.findMany()).length)) {
      await prisma.todo.createMany({ data: initialTodos })
    }

    console.log('Database has been successfully seeded 🚀')
  } catch (e) {
    console.log(e)
  } finally {
    await prisma.$disconnect()
  }
}

main()

```

В `package.json` определяем тип кода сервера (модуль), команды для запуска сервера в режиме для разработки и производственном режиме, а также команду для заполнения БД начальными данными:

```

"type": "module",
"scripts": {
  "dev": "nodemon",
  "start": "prisma generate && prisma migrate deploy && node index.js"
}

```

```
},  
"prisma": {  
  "seed": "node prisma/seed.js"  
}
```

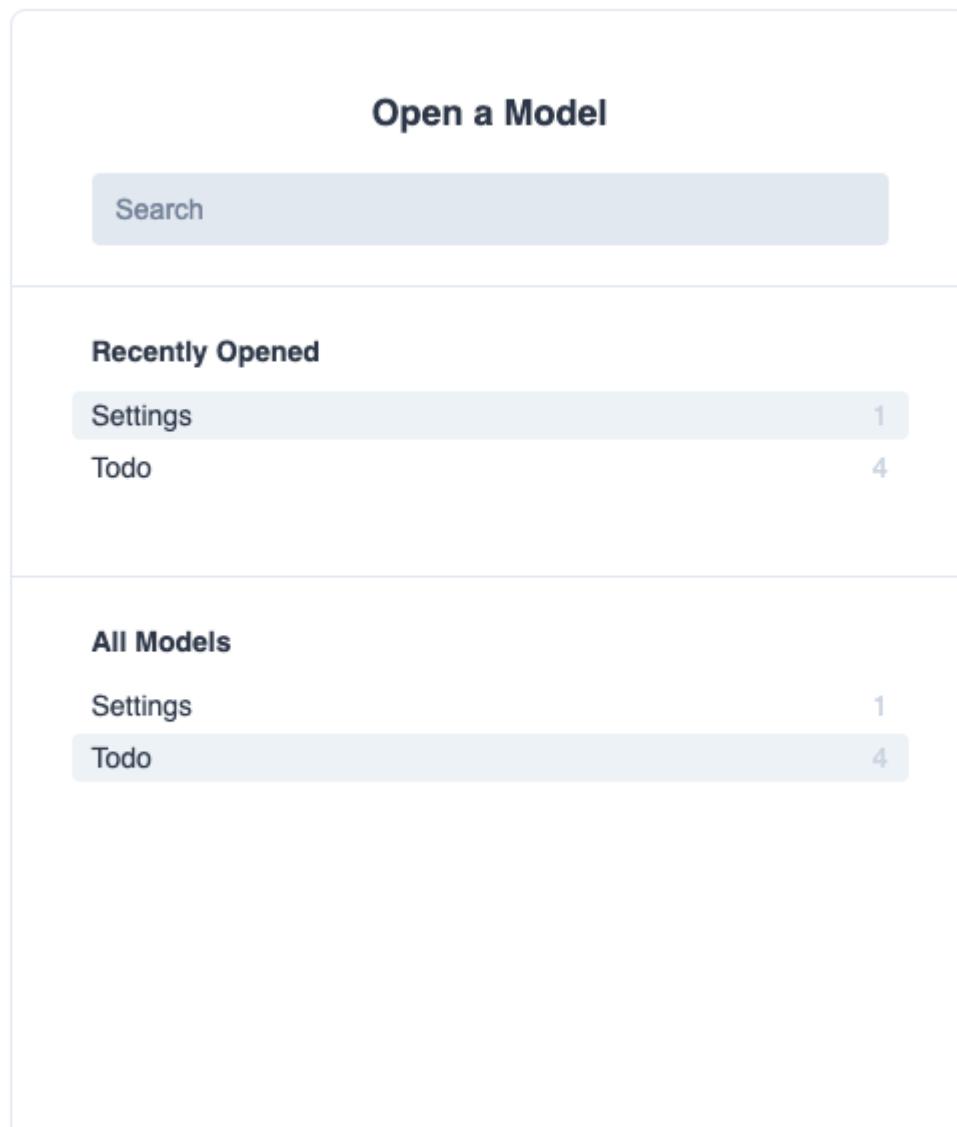
Заполняем БД начальными данными:

```
# мы находимся в директории `api`  
npx prisma db seed
```

Открываем нашу БД в интерактивном режиме:

```
npx prisma studio
```

Это приводит к открытию вкладки браузера по адресу <http://localhost:5555>:



Приступаем к разработке сервера.

Структура сервера будет следующей:

- routes
 - index.js
 - settings.routes.js - маршруты (роуты) для настроек
 - todo.routes.js - роуты для задач
- index.js

Содержание файла `index.js`:

```
// импортируем библиотеки и утилиты
import express from 'express'
import cors from 'cors'
import Prisma from '@prisma/client'
import apiRoutes from './routes/index.js'
import { join, dirname } from 'path'
import { fileURLToPath } from 'url'
const { PrismaClient } = Prisma

// создаем и экспортируем экземпляр 'prisma'
export const prisma = new PrismaClient()

// путь к текущей директории
const __dirname = dirname(fileURLToPath(import.meta.url))

// создаем экземпляр приложения 'express'
const app = express()

// отключаем 'cors'
app.use(cors())
// включаем парсинг 'json' в объекты
app.use(express.json())

// это пригодится нам при запуске приложения в производственном режиме
if (process.env.ENV === 'prod') {
    // обратите внимание на пути
    // путь к текущей директории + 'client/build'
    const clientBuildPath = join(__dirname, 'client', 'build')
    // путь к текущей директории + 'admin/dist'
    const adminDistPath = join(__dirname, 'admin', 'dist')
```

```
// обслуживание статических файлов
// клиент будет доступен по пути сервера
app.use(express.static(clientBuildPath))
app.use(express.static(adminDistPath))
// админка будет доступна по пути сервера + '/admin'
app.use('/admin', (req, res) => {
  res.sendFile(join(adminDistPath, decodeURIComponent(req.url)))
})
}

// роутинг
app.use('/api', apiRoutes)

// обработчик ошибок
app.use((err, req, res, next) => {
  console.log(err)
  const status = err.status || 500
  const message = err.message || 'Something went wrong. Try again later'
  res.status(status).json({ message })
})

// запускаем сервер на порту 5000
app.listen(5000, () => {
  console.log(`Server ready 🚀`)
})
```

Рассмотрим роуты.

Начнем с роутера приложения (`routes/index.js`):

```
import { Router } from 'express'
import todoRoutes from './todo.routes.js'
import settingsRoutes from './settings.routes.js'

const router = Router()

router.use('/todo', todoRoutes)
router.use('/settings', settingsRoutes)

export default router
```

Роутер для настроек (`routes/settings.routes.js`):

```

import { Router } from 'express'
import { prisma } from '../index.js'

const router = Router()

// получение настроек
router.get('/', async (req, res, next) => {
  try {
    const settings = await prisma.settings.findFirst()
    res.status(200).json(settings)
  } catch (e) {
    next(e)
  }
})

// обновление настроек
router.put('/:id', async (req, res, next) => {
  const id = Number(req.params.id)
  try {
    const settings = await prisma.settings.update({
      data: req.body,
      where: { id }
    })
    res.status(201).json(settings)
  } catch (e) {
    next(e)
  }
})

export default router

```

Роутер для задач (`routes/todo.routes.js`):

```

import { Router } from 'express'
import { prisma } from '../index.js'

const router = Router()

// получение задач
router.get('/', async (req, res, next) => {
  try {
    const todos = (await prisma.todo.findMany()).sort(
      (a, b) => a.created_at - b.created_at
    )
    res.status(200).json(todos)
  } catch (e) {
    next(e)
  }
})

export default router

```

```
)  
res.status(200).json(todos)  
} catch (e) {  
    next(e)  
}  
})  
  
// создание задачи  
router.post('/', async (req, res, next) => {  
    try {  
        const newTodo = await prisma.todo.create({  
            data: req.body  
        })  
        res.status(201).json(newTodo)  
    } catch (e) {  
        next(e)  
    }  
})  
  
// обновление задачи  
router.put('/:id', async (req, res, next) => {  
    const id = Number(req.params.id)  
    try {  
        const updatedTodo = await prisma.todo.update({  
            data: req.body,  
            where: { id }  
        })  
        res.status(201).json(updatedTodo)  
    } catch (e) {  
        next(e)  
    }  
})  
  
// удаление задачи  
router.delete('/:id', async (req, res, next) => {  
    const id = Number(req.params.id)  
    try {  
        await prisma.todo.delete({  
            where: { id }  
        })  
        res.sendStatus(201)  
    } catch (e) {  
        next(e)  
    }  
})
```

```
export default router
```

Это все, что требуется от нашего сервера.

Админка

Структура админки будет следующей (`admin/src`):

- components
 - App.vue - основной компонент приложения
 - Settings.vue - компонент для обновления настроек
- index.js

Начнем с основного компонента (`components/App.vue`).

Разметка:

```
<template>
  <div id="app">
    <h1>Admin</h1>
    <!-- Загрузка -->
    <h2 v-if="loading">Loading...</h2>
    <!-- Ошибка -->
    <h3 v-else-if="error" class="error">
      {{ error.message || 'Something went wrong. Try again later' }}
    </h3>
    <!-- Компонент для обновления настроек -->
    <div v-else>
      <h2>Settings</h2>
      <!-- Пропы: настройки, полученные от сервера (из БД), метод для их
получения и адрес API -->
      <Settings :settings="settings" :getSettings="getSettings" :apiUri="apiUri"
/>
    </div>
  </div>
</template>
```

Стили:

```
@import url('https://fonts.googleapis.com/css2?
family=Montserrat:wght@200;400;600&display=swap');

:root {
    --primary: #0275d8;
    --success: #5cb85c;
    --warning: #f0ad4e;
    --danger: #d9534f;
    --light: #f7f7f7;
    --dark: #292b2c;
}

* {
    font-family: 'Montserrat', sans-serif;
    font-size: 1rem;
}

body.light {
    background-color: var(--light);
    color: var(--dark);
}

body.dark {
    background-color: var(--dark);
    color: var(--light);
}

#app {
    display: flex;
    flex-direction: column;
    text-align: center;
}

h2 {
    font-size: 1.4rem;
}

form div {
    display: flex;
    flex-direction: column;
    align-items: center;
}

label {
```

```
margin: 0.5rem 0;  
}  
  
input {  
  padding: 0.5rem;  
  max-width: 220px;  
  width: max-content;  
  outline: none;  
  border: 1px solid var(--dark);  
  border-radius: 4px;  
  text-align: center;  
}  
  
input:focus {  
  border-color: var(--primary);  
}  
  
button {  
  margin: 1rem 0;  
  padding: 0.5rem 1rem;  
  background: none;  
  border: none;  
  border-radius: 4px;  
  outline: none;  
  background-color: var(--success);  
  color: var(--light);  
  box-shadow: 0 1px 2px rgba(0, 0, 0, 0.3);  
  cursor: pointer;  
  user-select: none;  
  transition: 0.2s;  
}  
  
button:active {  
  box-shadow: none;  
}  
  
.error {  
  color: var(--danger);  
}
```

Скрипт:

```
// импортируем компонент для обновления настроек  
import Settings from './Settings'
```

```
export default {
  // название компонента
  name: 'App',
  // дочерние компоненты
  components: {
    Settings
  },
  // начальное состояние
  data() {
    return {
      loading: true,
      error: null,
      settings: {},
      apiUri: 'http://localhost:5000/api/settings'
    }
  },
  // монтирование компонента
  created() {
    // получаем настройки
    this.getSettings()
  },
  // методы
  methods: {
    // для получения настроек
    async getSettings() {
      this.loading = true
      try {
        const response = await fetch(API_URI)
        if (!response.ok) throw response
        this.settings = await response.json()
      } catch (e) {
        this.error = e
      } finally {
        this.loading = false
      }
    }
  }
}
```

Теперь рассмотрим компонент для обновления настроек (`components/Settings.vue`).

Разметка:

```
<template>
  <!-- Загрузка -->
  <div v-if="loading">Loading...</div>
  <!-- Ошибка -->
  <div v-else-if="error">
    {{ error.message || JSON.stringify(error, null, 2) }}
  </div>
  <!-- Настройки -->
  <form v-else @submit.prevent="saveSettings">
    <!-- Приветствие -->
    <div>
      <label for="greetings">Greetings</label>
      <input
        type="text"
        id="greetings"
        name="greetings"
        :value="settings.greetings"
        required
      />
    </div>
    <!-- Тема -->
    <div>
      <label for="theme">Theme</label>
      <input
        type="text"
        id="theme"
        name="theme"
        :value="settings.theme"
        required
      />
    </div>
    <!-- Базовый размер шрифта -->
    <div>
      <label for="base_font_size">Base font size</label>
      <input
        type="text"
        id="base_font_size"
        name="base_font_size"
        :value="settings.base_font_size"
        required
      />
    </div>

    <button>Save</button>
```

```
</form>  
</template>
```

Скрипт:

```
export default {  
    // название компонента  
    name: 'Settings',  
    // пропы  
    props: {  
        settings: {  
            type: Object,  
            required: true  
        },  
        getSettings: {  
            type: Function,  
            required: true  
        },  
        apiUrl: {  
            type: String,  
            required: true  
        }  
    },  
    // начальное состояние  
    data() {  
        return {  
            loading: false,  
            error: null  
        }  
    },  
    // методы  
    methods: {  
        // для обновления настроек в БД  
        async saveSettings(e) {  
            this.loading = true  
            const formDataObj = [...new FormData(e.target)].reduce(  
                (obj, [key, val]) => ({  
                    ...obj,  
                    [key]: val  
                }),  
                {}  
            )  
            try {  
                const response = await fetch(`#${this.apiUrl}/${this.settings.id}`, {
```

```

        method: 'PUT',
        body: JSON.stringify(formDataObj),
        headers: {
          'Content-Type': 'application/json'
        }
      })
      if (!response.ok) throw response
      // получаем обновленные настройки
      await this.getSettings()
    } catch (e) {
      this.error = e
    } finally {
      this.loading = false
    }
  }
}

```

На этом с админкой мы закончили.

Клиент

Структура клиента будет следующей (`client/src`):

- api
 - settings.api.js - API для настроек
 - todo.api.js - API для задач
- components
 - TodoForm.js - компонент для создания задачи
 - TodoList.js - компонент для формирования списка задач
- hooks
 - useStore.js - хранилище состояния в виде пользовательского хука
- App.js - основной компонент приложения
- App.css
- index.js

Для управления состоянием приложения мы будем использовать `Zustand`.

Устанавливаем его:

```
yarn add zustand
```

Начнем с API для настроек (`api/settings.api.js`):

```
// конечная точка
const API_URI = 'http://localhost:5000/api/settings'

// метод для получения настроек
const fetchSettings = async () => {
  try {
    const response = await fetch(API_URI)
    if (!response.ok) throw response
    return await response.json()
  } catch (e) {
    throw e
  }
}

const settingsApi = { fetchSettings }

export default settingsApi
```

API для задач (`api/todo.api.js`):

```
// конечная точка
const API_URI = 'http://localhost:5000/api/todo'

// метод для получения задач
const fetchTodos = async () => {
  try {
    const response = await fetch(API_URI)
    if (!response.ok) throw response
    return await response.json()
  } catch (e) {
    throw e
  }
}

// метод для создания новой задачи
const addTodo = async (newTodo) => {
  try {
    const response = await fetch(API_URI, {
      method: 'POST',
      body: JSON.stringify(newTodo),
      headers: {
```

```

        'Content-Type': 'application/json'
    }
})
if (!response.ok) throw response
return await response.json()
} catch (e) {
    throw e
}
}

// метод для обновления задачи
const updateTodo = async (id, changes) => {
try {
    const response = await fetch(`#${API_URI}/${id}`, {
        method: 'PUT',
        body: JSON.stringify(changes),
        headers: {
            'Content-Type': 'application/json'
        }
    })
    if (!response.ok) throw response
    return await response.json()
} catch (e) {
    throw e
}
}

// метод для удаления задачи
const removeTodo = async (id) => {
try {
    const response = await fetch(`#${API_URI}/${id}`, {
        method: 'DELETE'
    })
    if (!response.ok) throw response
} catch (e) {
    throw e
}
}

const todoApi = { fetchTodos, addTodo, updateTodo, removeTodo }

export default todoApi

```

Хранилище состояния в виде пользовательского хука (`hooks/useStore.js`):

```
import create from 'zustand'
// API для настроек
import settingsApi from '../api/settings.api'
// API для задач
import todoApi from '../api/todo.api'

const useStore = create((set, get) => ({
  // начальное состояние
  settings: {},
  todos: [],
  loading: false,
  error: null,
  // методы для
  // получения настроек
  fetchSettings() {
    set({ loading: true })
    settingsApi
      .fetchSettings()
      .then(settings) => {
        set({ settings })
      }
      .catch(error) => {
        set({ error })
      }
      .finally(() => {
        set({ loading: false })
      })
  },
  // получения задач
  fetchTodos() {
    set({ loading: true })
    todoApi
      .fetchTodos()
      .then(todos) => {
        set({ todos })
      }
      .catch(error) => {
        set({ error })
      }
      .finally(() => {
        set({ loading: false })
      })
  },
  // создания задачи
})
```

```
addTodo(newTodo) {
  set({ loading: true })
  todoApi
    .addTodo(newTodo)
    .then((newTodo) => {
      const todos = [...get().todos, newTodo]
      set({ todos })
    })
    .catch((error) => {
      set({ error })
    })
    .finally(() => {
      set({ loading: false })
    })
},
// обновления задачи
updateTodo(id, changes) {
  set({ loading: true })
  todoApi
    .updateTodo(id, changes)
    .then((updatedTodo) => {
      const todos = get().todos.map((todo) =>
        todo.id === updatedTodo.id ? updatedTodo : todo
      )
      set({ todos })
    })
    .catch((error) => {
      set({ error })
    })
    .finally(() => {
      set({ loading: false })
    })
},
// удаления задачи
removeTodo(id) {
  set({ loading: true })
  todoApi
    .removeTodo(id)
    .then(() => {
      const todos = get().todos.filter((todo) => todo.id !== id)
      set({ todos })
    })
    .catch((error) => {
      set({ error })
    })
}
```

```

    .finally(() => {
      set({ loading: false })
    })
  })
}

export default useStore

```

Компонент для создания новой задачи (`components/TodoForm.js`):

```

import { useState, useEffect } from 'react'
import useStore from '../hooks/useStore'

export default function TodoForm() {
  // метод для создания задачи из хранилища
  const addTodo = useStore(({ addTodo }) => addTodo)
  // состояние для текста новой задачи
  const [text, setText] = useState('')
  const [disable, setDisable] = useState(true)

  useEffect(() => {
    setDisable(!text.trim())
  }, [text])

  // метод для обновления текста задачи
  const onChange = ({ target: { value } }) => {
    setText(value)
  }

  // метод для отправки формы
  const onSubmit = (e) => {
    e.preventDefault()
    if (disable) return
    const newTodo = {
      text,
      done: false
    }
    addTodo(newTodo)
  }

  return (
    <form onSubmit={onSubmit}>
      <label htmlFor='text'>New todo text</label>
      <input type='text' id='text' value={text} onChange={onChange} />
    
```

```

    <button className='add'>Add</button>
  </form>
)
}

```

Компонент для формирования списка задач (`components/TodoList.js`):

```

import useStore from '../hooks/useStore'

export default function TodoList() {
  // задачи и методы для обновления и удаления задачи из хранилища
  const { todos, updateTodo, removeTodo } = useStore(
    ({ todos, updateTodo, removeTodo }) => ({ todos, updateTodo, removeTodo })
  )

  return (
    <ul>
      {todos.map(({ id, text, done }) => (
        <li key={id}>
          <input
            type='checkbox'
            checked={done}
            onChange={() => {
              updateTodo(id, { done: !done })
            }}
          />
          <span>{text}</span>
          <button
            onClick={() => {
              removeTodo(id)
            }}
          >
            Remove
          </button>
        </li>
      ))}
    </ul>
  )
}

```

Основной компонент приложения (`App.js`):

```
import { useEffect } from 'react'
import './App.css'
import useStore from './hooks/useStore'
import TodoForm from './components/TodoForm'
import TodoList from './components/TodoList'

// получаем настройки
useStore.getState().fetchSettings()
// получаем задачи
useStore.getState().fetchTodos()

function App() {
  // настройки, индикатор загрузки и ошибка из хранилища
  const { settings, loading, error } = useStore(
    ({ settings, loading, error }) => ({ settings, loading, error })
  )

  useEffect(() => {
    if (Object.keys(settings).length) {
      // применяем базовый размер шрифта к элементу 'html'
      document.documentElement.style.fontSize = settings.base_font_size
      // применяем тему
      document.body.className = settings.theme
    }
  }, [settings])

  // загрузка
  if (loading) return <h2>Loading...</h2>

  // ошибка
  if (error)
    return (
      <h3 className='error'>
        {error.message || 'Something went wrong. Try again later'}
      </h3>
    )

  return (
    <div className='App'>
      <h1>Client</h1>
      <h2>{settings.greetings}</h2>
      <TodoForm />
      <TodoList />
    </div>
  )
}
```

```
)  
}  
  
export default App
```

Стили (`App.css`):

```
@import url('https://fonts.googleapis.com/css2?  
family=Montserrat:wght@200;400;600&display=swap');
```

```
:root {  
    --primary: #0275d8;  
    --success: #5cb85c;  
    --warning: #f0ad4e;  
    --danger: #d9534f;  
    --light: #f7f7f7;  
    --dark: #292b2c;  
}  
  
* {  
    margin: 0;  
    padding: 0;  
    box-sizing: border-box;  
    font-family: 'Montserrat', sans-serif;  
    font-size: 1rem;  
}  
  
/* Тема */  
body.light {  
    background-color: var(--light);  
    color: var(--dark);  
}  
  
body.dark {  
    background-color: var(--dark);  
    color: var(--light);  
}  
/* --- */  
  
#root {  
    padding: 1rem;  
    display: flex;  
    justify-content: center;  
}
```

```
.App {  
  display: flex;  
  flex-direction: column;  
  align-items: center;  
}  
  
h1,  
h2 {  
  margin: 1rem 0;  
}  
  
h1 {  
  font-size: 1.6rem;  
}  
  
h2 {  
  font-size: 1.4rem;  
}  
  
h3 {  
  font-size: 1.2rem;  
}  
  
label {  
  margin-bottom: 0.5rem;  
  display: block;  
}  
  
form {  
  margin: 1rem 0;  
}  
  
form input {  
  padding: 0.5rem;  
  max-width: 220px;  
  width: max-content;  
  outline: none;  
  border: 1px solid var(--dark);  
  border-radius: 4px;  
  text-align: center;  
}  
  
form input:focus {  
  border-color: var(--primary);  
}
```

```
}

ul {
    list-style: none;
}

li {
    margin: 0.75rem 0;
    display: flex;
    align-items: center;
    justify-content: space-between;
}

li input {
    width: 18px;
    height: 18px;
}

li span {
    display: block;
    width: 120px;
    word-break: break-all;
}

button {
    padding: 0.5rem 1rem;
    background: none;
    border: none;
    border-radius: 4px;
    outline: none;
    background-color: var(--danger);
    color: var(--light);
    box-shadow: 0 1px 2px rgba(0, 0, 0, 0.3);
    cursor: pointer;
    user-select: none;
    transition: 0.2s;
}

button:active {
    box-shadow: none;
}

button.add {
    background-color: var(--success);
}
```

```
.error {
  color: var(--danger);
}

.App {
  text-align: center;
}
```

Поскольку отступы и размеры заданы с помощью `гем`, мы легко можем манипулировать этими значениями, меняя размер шрифта элемента `html`.

На этом с клиентом мы также закончили.

Проверка работоспособности приложения

Поднимаемся в корневую директорию (`docker-test`), инициализируем `Node.js-проект` и устанавливаем `concurrently` - утилиту для одновременного выполнения команд, определенных в файле `package.json`:

```
# мы находимся в директории `docker-test`
yarn init -yр
yarn add concurrently
```

Определяем команды для запуска серверов для разработки в `package.json`:

```
"scripts": {
  "dev:client": "yarn --cwd services/client start",
  "dev:admin": "yarn --cwd services/admin dev",
  "dev:api": "yarn --cwd services/api dev",
  "dev": "concurrently \"yarn dev:client\" \"yarn dev:admin\" \"yarn dev:api\""
}
```

Выполняем команду `yarn dev` или `npm run dev`.

Это приводит к запуску 3 серверов для разработки:

- для клиента по адресу `http://localhost:3000`:

Client

Welcome to Docker Test App

New todo text

Add

<input checked="" type="checkbox"/>	Eat	Remove
<input checked="" type="checkbox"/>	Code	Remove
<input type="checkbox"/>	Sleep	Remove
<input type="checkbox"/>	Repeat	Remove

- для админки по адресу `http://localhost:4000`:

Admin

Settings

Greetings

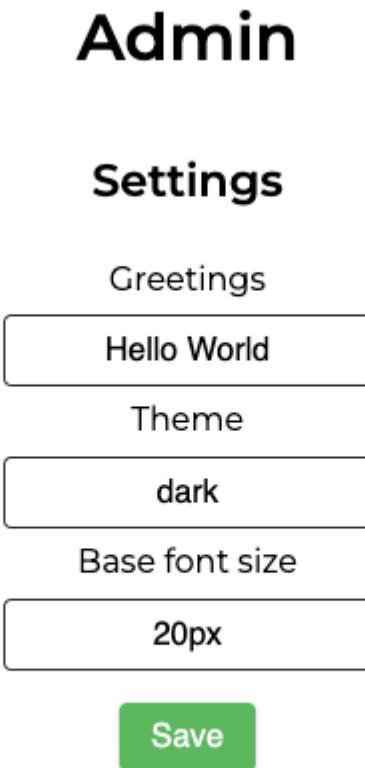
Theme

Base font size

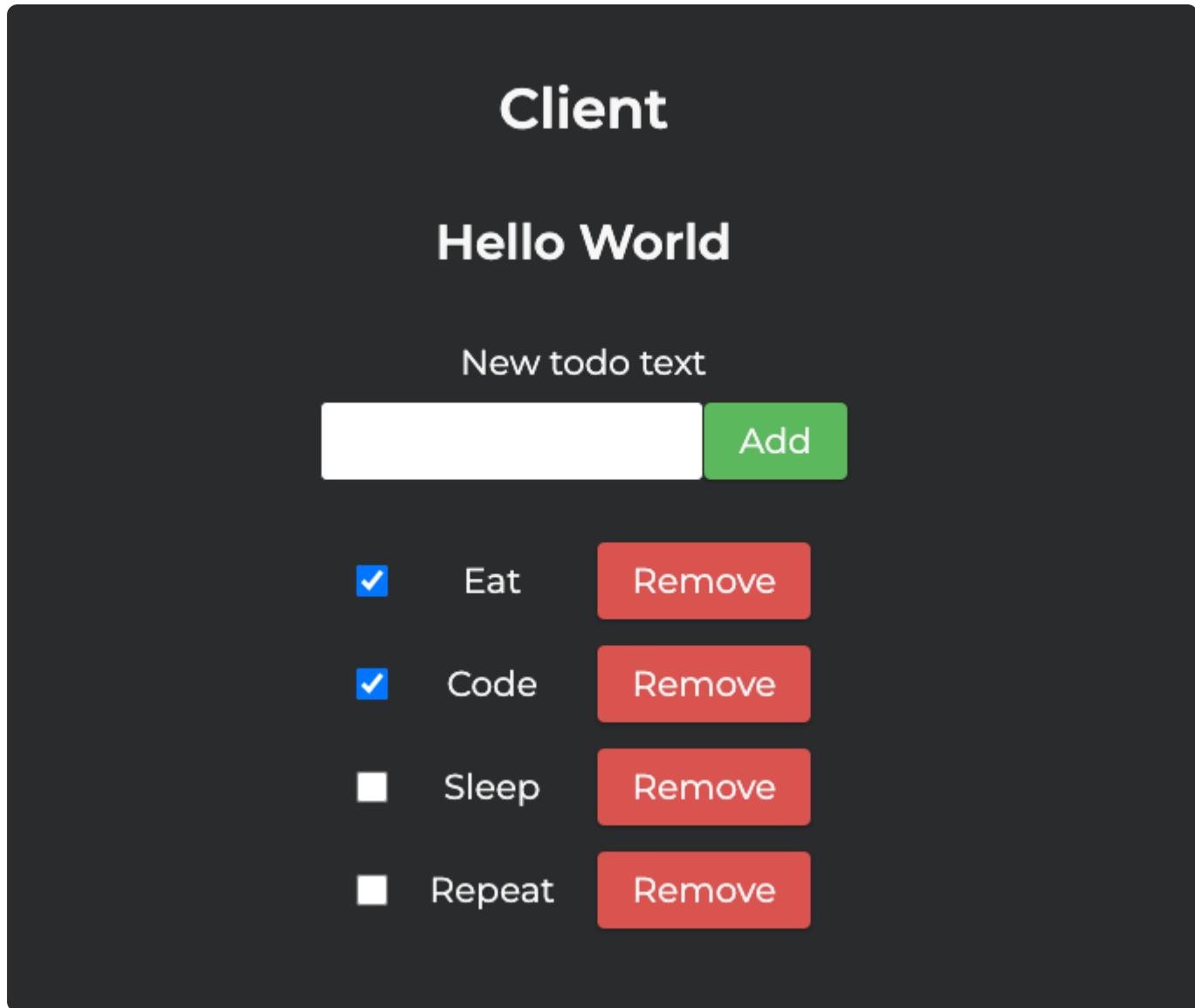
Save

- для сервера по адресу `http://localhost:5000`.

Меняем настройки в админке:

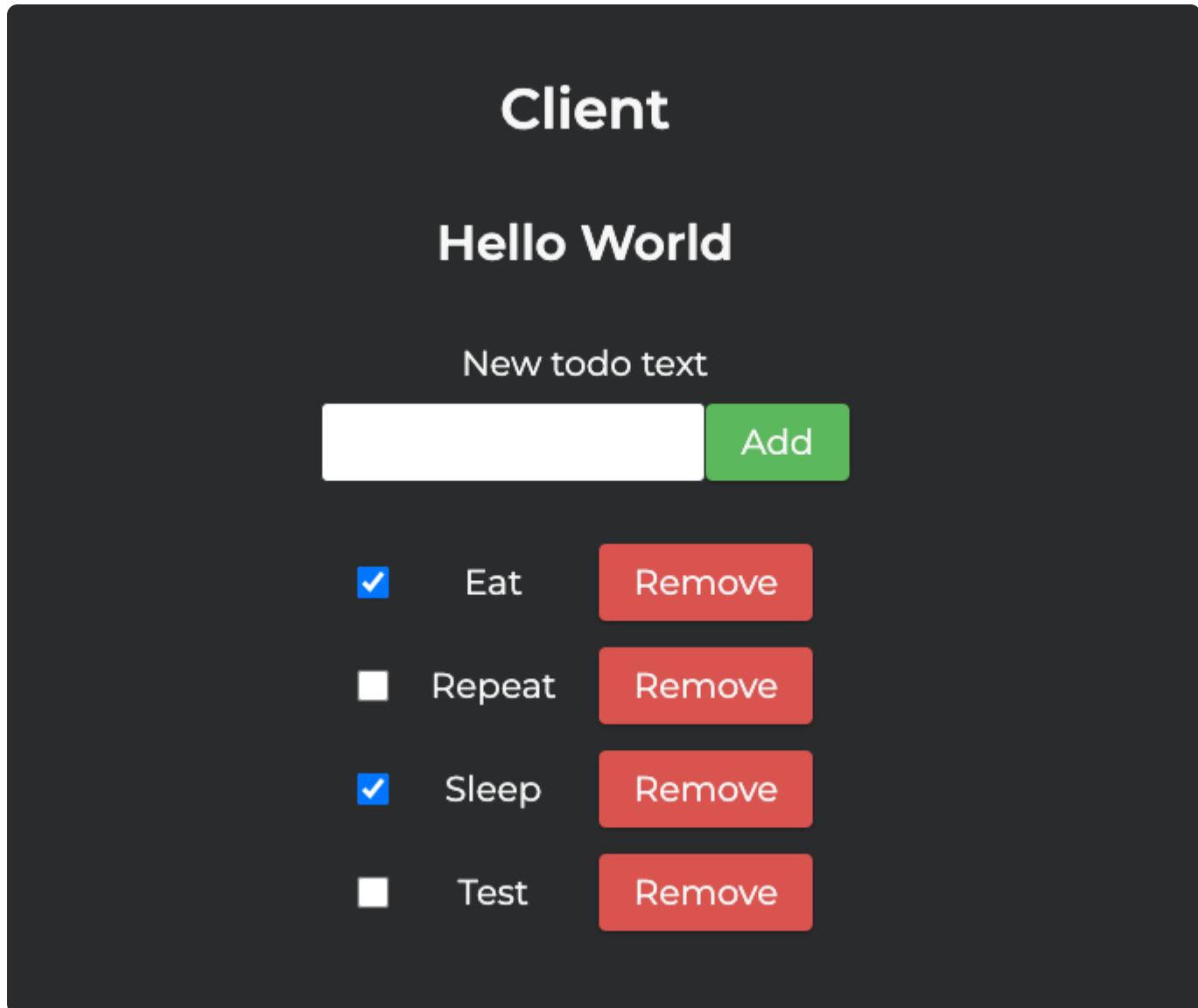


Перезагружаем клиента:



Видим, что настройки успешно применились.

Работаем с задачами:



Задачи успешно создаются/обновляются/удаляются и сохраняются в БД.

Отлично. Приложение работает, как ожидается.

[Репозиторий с кодом приложения .](#)

Если вы используете `npm`, команды для запуска серверов для разработки в файле `package.json` должны выглядеть так:

```
"scripts": {  
  "dev:client": "npm run start --prefix services/client",  
  "dev:admin": "npm run dev --prefix services/admin",  
  "dev:api": "npm run dev --prefix services/api",  
  "dev": "concurrently \"npm run dev:client\" \"npm run dev:admin\" \"npm run dev:api\""  
}
```

"Контейнеризация" приложения

Dockerfile

Начнем с определения **Dockerfile** для сервисов нашего приложения.

В директории **client** создаем файл **Dockerfile** следующего содержания:

```
# дефолтная версия 'Node.js'  
ARG NODE_VERSION=16.13.1  
  
# используемый образ  
FROM node:$NODE_VERSION  
  
# рабочая директория  
WORKDIR /client  
  
# копируем указанные файлы в рабочую директорию  
COPY package.json yarn.lock ./  
  
# устанавливаем зависимости  
RUN yarn  
  
# копируем остальные файлы  
COPY . .  
  
# выполняем сборку приложения  
RUN yarn build
```

Обратите внимание: на данном этапе вместо сборки (**RUN yarn build**) мы могли бы выполнять команду **start** для запуска сервера для разработки: **CMD ["yarn", "start"]**, но если мы так сделаем, то впоследствии нам придется создавать отдельный **Dockerfile** для продакшна. Пуще сразу определить производственную версию **Dockerfile**, а команду **start** запускать из **docker-compose.yml**.

Создаем практически идентичный **Dockerfile** в директории **admin**:

```
ARG NODE_VERSION=16.13.1  
  
FROM node:$NODE_VERSION as build
```

```
WORKDIR /admin

COPY package.json yarn.lock ./

RUN yarn

COPY . .

RUN yarn build
```

Обратите внимание: сборка клиента будет находиться в директории `client/build`, а сборка админки - в директории `admin/dist`. В файле `api/index.js` можно найти такие строки:

```
if (process.env.ENV === 'production') {
  const clientBuildPath = join(__dirname, 'client', 'build')
  const adminDistPath = join(__dirname, 'admin', 'dist')

  app.use(express.static(clientBuildPath))
  app.use(express.static(adminDistPath))
  app.use('/admin', (req, res) => {
    res.sendFile(join(adminDistPath, decodeURIComponent(req.url)))
  })
}
```

Эти строки говорят нам о том, что при запуске сервера в производственном режиме (`process.env.ENV === 'production'`), он будет обслуживать статические файлы из названных выше директорий: клиент будет доступен по маршруту (роуту) `/`, а админка - по роуту `/admin`. Мы вернемся к этому позже.

Создаем похожий `Dockerfile` в директории `api`:

```
ARG NODE_VERSION=16.13.1

FROM node:$NODE_VERSION

WORKDIR /app

COPY package.json yarn.lock ./

RUN yarn
```

```
COPY . .

# выставляем порт
EXPOSE 5000

# запускаем сервер в производственном режиме
CMD ["yarn", "start"]
```

Обратите внимание: инструкции `EXPOSE 5000` и `CMD ["yarn", "start"]` на данном этапе можно опустить, но они потребуются нам в продакшне. На самом деле, нам потребуется кое-что еще, но позвольте пока сохранить интригу.

Также *обратите внимание*, что я внес парочку изменений в проект:

- Содержание файла `.env`, находящегося корневой директории проекта:

```
# добавил название приложения
APP_NAME=my-app

# уточнил версию `Node.js`
NODE_VERSION=16.13.1

POSTGRES_VERSION=14
POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres
POSTGRES_DB=mydb

# перенес сюда путь к БД из файла `api/.env`
# обратите внимание, что вместо `localhost` после символа `@` мы указываем
название контейнера - `postgres`
DATABASE_URL=postgresql://postgres:postgres@postgres:5432/mydb?schema=public

ENV=development
```

- Команда для запуска сервера для разработки (файл `api/package.json`, раздел `scripts`):

```
"dev": "prisma migrate dev && prisma db seed && nodemon",
```

Хорошей практикой считается исключение файлов из образа с помощью `.dockerignore`:

```
node_modules  
yarn-error.log  
# mac  
.DS_Store
```

Такой файл нужно создать в каждом сервисе.

После создания `Dockerfile` для каждого сервиса мы готовы к "контейнеризации" приложения.

Docker Compose

Создаем в корневой директории файл `docker-compose.dev.yml` следующего содержания:

```
# версия 'compose'  
version: '3.9'  
# сервисы  
services:  
  # БД  
  postgres:  
    # файл, содержащий переменные среды окружения  
    env_file: .env  
    # название контейнера  
    container_name: ${APP_NAME}_postgres  
    # используемый образ  
    image: postgres:${POSTGRES_VERSION}  
    # именованный том для хранения данных  
    volumes:  
      - data_postgres:/var/lib/postgresql/data  
    # порт для доступа к БД  
    ports:  
      - 5432:5432  
    # политика перезапуска контейнера  
    restart: on-failure  
  
  client:  
    env_file: .env  
    container_name: ${APP_NAME}_client  
    image: node:${NODE_VERSION}  
    # рабочая директория  
    working_dir: /app  
    # анонимный том
```

```

# 'rw' означает 'read/write' - чтение/запись
volumes:
  - ./services/client:/app:rw
# сервис, от которого зависит работоспособность данного сервиса
depends_on:
  - api
ports:
  - 3000:3000
restart: on-failure
# команда для запуска сервера для разработки
command: bash -c "yarn start"

admin:
  env_file: .env
  container_name: ${APP_NAME}_admin
  image: node:${NODE_VERSION}
  working_dir: /app
  volumes:
    - ./services/admin:/app:rw
  depends_on:
    - api
  ports:
    - 4000:4000
  restart: on-failure
  command: bash -c "yarn dev"

api:
  env_file: .env
  container_name: ${APP_NAME}_api
  # ссылка на 'Dockerfile', на основе которого выполняется сборка
  build: services/api
  ports:
    - 5000:5000
  depends_on:
    - postgres
  restart: on-failure
  # перезапись команды 'yarn start', определенной в 'Dockerfile'
  command: bash -c "yarn dev"

# тома
volumes:
  data_postgres:

```

Определим в `package.json` несколько команд для управления `compose`:

```
"dev:compose:up": "docker compose -f docker-compose.dev.yml up -d",
"dev:compose:stop": "docker compose -f docker-compose.dev.yml stop",
"dev:compose:rm": "docker compose -f docker-compose.dev.yml rm",
"compose:up": "docker compose up -d",
"compose:stop": "docker compose stop",
"compose:rm": "docker compose rm"
```

Команда `compose:up` поднимает, команда `compose:stop` - останавливает, а команда `compose:rm` - удаляет сервис. Префикс `dev:` означает что поднимается/останавливается/удаляется сервис для разработки. В свою очередь, отсутствие данного префикса означает управление производственным сервисом (по умолчанию `compose` использует файл `docker-compose.yml`, которым мы займемся позже).

Еще несколько команд, которые могут пригодится при работе с `compose` при отладке приложения:

```
# список запущенных контейнеров
docker ps

# список запущенных сервисов
docker compose ls

# список образов
docker images

# удаление образа
# [image-name] - название образа
docker image rm [image-name]
# например
docker image rm docker-test_api

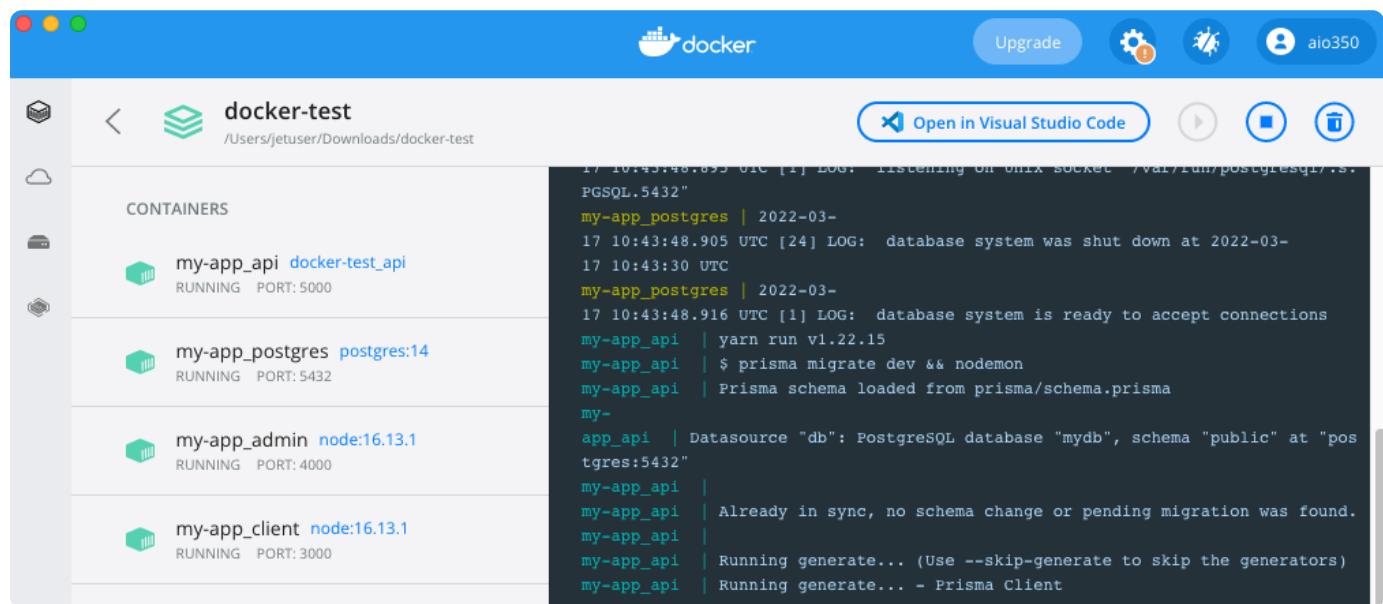
# список томов
docker volume ls

# удаление тома
# [volume-name] - название тома
docker volume rm [volume-name]
# например
docker volume rm postgres_data
```

```
# очистка системы (т тома не удаляются)
docker system prune -a
```

Поднимаем сервис в режиме для разработки с помощью команды `yarn dev:compose:up` или `npm run dev:compose:up`:

```
[jetuser@JET100067-1937 docker-test % yarn dev:compose:up
yarn run v1.22.17
$ docker compose -f docker-compose.dev.yml up -d
[+] Running 4/4
# Container my-app_postgres Started
# Container my-app_api Started
# Container my-app_admin-test Started
# Container my-app_client Started
Done in 9.42s
```



После создания контейнеров сервисам потребуется какое-то время на запуск, после чего они будут доступны по следующим адресам:

- клиент: `localhost:3000`;
- админка: `localhost:4000`;
- сервер: `localhost:5000` (нет прямого доступа; доступен для клиента и админки);
- БД: `postgres:5432` (нет прямого доступа; доступен только для сервера).

По сути, команда `dev:compose:up` делает то же самое, что и команда `dev` + скрипт из файла `db`.

Чем производственный сервис будет отличаться от сервиса для разработки? Предположим, что мы хотим, чтобы всю статику приложения обслуживал сервер, поэтому нам требуется какой-то способ передать `api` сборки клиента и админки. Существует несколько способов это сделать. Одним из самых простых и удобных является использование [Docker Hub](#).

Переходим по ссылке и создаем аккаунт.

Переходим в директорию `client` и создаем образ с тегом:

```
cd client
# [username] - ваш логин для входа в dockerhub
# тег образа обязательно должен начинаться с вашего логина
docker build . -t [username]/docker-test_client
# мой логин - aio350
docker build . -t aio350/docker-test_client
```

Авторизуемся в `dockerhub` и отправляем образ в свой реестр:

```
docker login
docker push aio350/docker-test_client
```

Делаем то же самое для админки:

```
cd admin
docker build . -t aio350/docker-test_admin
docker push aio350/docker-test_admin
```

После этого в своем реестре `dockerhub` мы увидим следующую картину:

Repository	Last pushed	Status	Stars	Downloads	Visibility
aio350 / docker-test_admin	a day ago	Not Scanned	0	0	Public
aio350 / docker-test_client	a day ago	Not Scanned	0	0	Public

Немного отредактируем файл `api/Dockerfile`:

```
ARG NODE_VERSION=16.13.1
```

```

# копируем образ клиента из `dockerhub`
# `AS` позволяет ссылаться на этот слой в других инструкциях
FROM aio350/docker-test_client AS client
# образ админки
FROM aio350/docker-test_admin AS admin

FROM node:$NODE_VERSION

WORKDIR /app

COPY package.json yarn.lock ./

RUN yarn

COPY . .

# копируем сборку клиента
COPY --from=client /client/build /app/client/build
# копируем сборку админки
COPY --from=admin /admin/dist /app/admin/dist

EXPOSE 5000

CMD ["yarn", "start"]

```

Создаем в корневой директории проекта файл `docker-compose.yml` следующего содержания:

```

version: '3.9'
services:
  postgres:
    env_file: .env
    container_name: ${APP_NAME}_postgres
    image: postgres:${POSTGRES_VERSION}
    volumes:
      - data_postgres:/var/lib/postgresql/data
    ports:
      - 5432:5432
    restart: on-failure
  # статика нашего приложения обслуживается сервером
  # поэтому нам не нужно поднимать сервисы 'client' и 'admin'
  api:
    env_file: .env
    # перезаписываем переменную 'ENV', определенную в файле '.env'

```

```

environment:
  - ENV=production
container_name: ${APP_NAME}_api
build: services/api
depends_on:
  - postgres
ports:
  - 5000:5000
restart: on-failure
# выполняется команда 'yarn start', определенная в 'Dockerfile'

volumes:
  data_postgres:

```

Удаляем сервис для разработки, удаляем образ `docker-test_api`, удаляем том `docker-test_data_postgres` и поднимаем производственный сервис:

```

yarn dev:compose:stop
yarn dev:compose:rm

# для чистоты эксперимента
docker image rm docker-test_api

docker volume rm docker-test_data_postgres

yarn compose:up

```

```
[jetuser@JET100067-1937 docker-test % yarn compose:up
yarn run v1.22.17
$ docker compose up -d
[+] Running 2/2
  # Container my-app_postgres  Started
  # Container my-app_api    Started
Done in 2.62s.
```

Теперь наш сервис состоит всего из 2 контейнеров.

Клиент доступен по адресу: `localhost:5000`, а админка - по адресу `localhost:5000/admin`.

Client

Welcome to Docker Test App

New todo text

Add

<input checked="" type="checkbox"/>	Eat	Remove
<input checked="" type="checkbox"/>	Code	Remove
<input type="checkbox"/>	Sleep	Remove
<input type="checkbox"/>	Repeat	Remove

Admin

Settings

Greetings

Welcome to Docker Test

Theme

light

Base font size

16px

Save

Приложение работает, как ожидается.

На этом "контейнеризацию" нашего приложения можно считать завершенной.

Tags:

docker docker compose devops container guide руководство контейнер

 Edit this page