Лабораторная работа №7 (DevOps №3). Система контейнеризации Docker, управление контейнерами Docker Compose

Оглавление

Curl

cURL - (распространяемая по лицензии MIT) кроссплатформенная служебная программа командной строки, позволяющая взаимодействовать с множеством различных серверов по множеству различных протоколов с синтаксисом URL.

Программа **cURL** может автоматизировать передачу файлов или последовательность таких операций. Например, это хорошее средство для моделирования действий пользователя в веб-обозревателе.

Команда *curl* предназначена для передачи данных по сети без взаимодействия с пользователем.

Программа поддерживает протоколы: FTP, FTPS, HTTP, HTTPS, TFTP, SCP, SFTP, Telnet, DICT, LDAP, а также POP3, IMAP и SMTP. Также cURL поддерживает сертификаты HTTPS, методы HTTP POST, HTTP PUT, загрузку на FTP, загрузку через формы HTTP.

Синтаксис

curl [опции...] <url>

Опции:

-d, --data <data>	Данные HTTP POST
-f, --fail	Быстрый сбой без вывода ошибок
-h, --help <category>	Получить помощь по командам

-i, --include	Включить заголовки ответа протокола в выходные данные
-o, --output <file>	Запись в файл
-O, --remote-name	Запись вывода в файл с именем удаленного файла
-s, --silent	Безмолвный режим
-T, --upload-file <file>	Передать локальный ФАЙЛ в пункт назначения
-u, --user <user:password>	Пользователь сервера и пароль
-A, --user-agent <name>	Отправить User-Agent <имя> на сервер
-v, --verbose	Сделать операцию более разговорчивой
-V, --version	Показать номер версии и выйти

Установка

Проверьте, установлен ли curl в вашей системе:

```
curl

curl: try 'curl --help' for more information
```

При его отсутствии будет получено следующее:

```
curl command not found
```

Установка curl в Ubuntu:

```
sudo apt install curl
```

Примеры использования команды curl в Linux

Возврат содержимого домашней страницы по url-адресу:

```
curl example.com
```

Скачивание файла с сохранением под оригинальным именем:

```
curl -O https://slackbuilds.org/slackbuilds/15.0/games/extremetuxracer.tar.gz
```

Скачивание файла с сохранением под другим именем:

```
curl -o etr.tar.gz https://slackbuilds.org/slackbuilds/15.0/games/extremetuxracer.tar.gz
```

Для того чтобы в случае обрыва соединения при скачивании большого файла не пришлось его закачивать заново следует использовать опцию **-O**. Она позволит продолжить загрузку файла.

Если вы не хотите, чтобы **curl** занял всю пропускную способность канала, то можно ограничить скорость скачивания.

Ограничение скорости до 250K:

```
curl --limit-rate 250K https://releases.ubuntu.com/22.04/ubuntu-22.04-desktop-amd64.iso22.04/ubuntu-22.04-desktop-amd64.iso -O
```

В случае защищённого FTP-сервера используется опция **-u** и указываются имя пользователя и пароль:

```
curl -u FTP_USERNAME:FTP_PASSWORD ftp://ftp.protection.com/
```

После входа в систему в домашнем каталоге пользователя выводится список файлов и каталогов.

Для скачивания одного из файлов выполняется следующая команда

```
curl -u FTP_USERNAME:FTP_PASSWORD ftp://ftp.protection.com/file-1.tar.gz
```

Для загрузки файла на сервер используется опция **-T**:

```
curl -T file-new.tar.gz -u FTP_USERNAME:FTP_PASSWORD ftp://ftp.protection.com/
```

nginx

Nginx (engine x — по-русски произносится как энджи́нкс или э́нжин-и́кс) — веб-сервер и почтовый прокси-сервер, работающий на Unix-подобных операционных системах (тестировалась сборка и работа на FreeBSD, OpenBSD, Linux, Solaris, macOS, AIX и HP-UX).

Nginx позиционируется производителем как простой, быстрый и надёжный сервер, не перегруженный функциями.

Применение **nginx** целесообразно прежде всего для статических веб-сайтов и как обратного прокси-сервера перед динамическими сайтами.

HTTP-сервер

- обслуживание неизменяемых запросов, индексных файлов, автоматическое создание списка файлов, кэш дескрипторов открытых файлов
- акселерированное проксирование без кэширования, простое распределение нагрузки и отказоустойчивость
- поддержка кеширования при акселерированном проксировании и FastCGI
- акселерированная поддержка FastCGI и memcached-серверов, простое распределение нагрузки и отказоустойчивость
- модульность, фильтры, в том числе сжатие (gzip), byte-ranges (докачка), chunked-ответы, HTTP-аутентификация, SSI-фильтр
- несколько подзапросов на одной странице, обрабатываемых в SSI-фильтре через прокси или FastCGI, выполняются параллельно

- поддержка SSL
- поддержка PSGI, WSGI
- экспериментальная поддержка встроенного Perl

SMTP/IMAP/POP3-прокси сервер

- перенаправление пользователя на SMTP/IMAP/POP3-бэкенд с использованием внешнего HTTP-сервера аутентификации
- простая аутентификация (LOGIN, USER/PASS)
- поддержка SSL и STARTTLS

По данным Netcraft на август 2020 года, число сайтов, обслуживаемых nginx, превышает 448 миллионов, что делает его первым по популярности веб-сервером в мире. Доля среди активных сайтов — 19,74 %, что ставит **nginx** на второе место после веб-сервера **Apache**.

По данным W3Techs, **nginx** наиболее часто используется на высоконагруженных сайтах, занимая первое место по частоте использования среди 100 000 самых посещаемых сайтов в мире — больше трети из них работает на **nginx**.

По данным российского регистратора REG.RU, **nginx** является самым популярным веб-сервером доменных зон .ru, .рф и .su, обслуживая более половины каждого сегмента. **nginx** — самый популярный веб-сервер в России с долей рынка 65,90 %.

Среди известных проектов, использующих nginx: Рамблер, Яндекс, ВКонтакте, Facebook, Netflix, Instagram, Mail.ru, Хабр, Живой Журнал, Avito.ru, Badoo, Wordpress.com, SourceForge.net, Qiwi.com, Groupon, Pinterest, Tumblr, Superjob.ru, HeadHunter, 2ГИС и многие другие.

Создание собственной html-страницы

По умолчанию, страница по умолчанию, отдаваемая запущенных nginx-сервером, располагается в Linux по адресу:

```
/var/www/html/index.html
```

Можно заменить эту страницу, либо использовать виртуальный хост по другому пути.

Виртуальный хост – это метод, который позволяет размещать несколько доменных имен на одном сервере.

Создадим простую html-страницу в директории **/var/www/tutorial/** (можно указывать любую), в ней создадим **index.html**.

Пример содержимого **index.html**:

```
<!doctype html>
<html>
```

```
<head>
   <meta charset="utf-8">
   <title>Hello, Nginx!</title>
</head>
<body>
   <h1>Hello, Nginx!</h1>
   <p>We have just configured our Nginx web server on Ubuntu Server!</p>
</body>
</html>
```

Сохраним файл.

Настройка виртуального хоста

Для настройки виртуального хоста нужно создать файл в директории **/etc/nginx/sites-enabled/**

Для примера сделаем сайт доступных на 81 порту вместо стандартного 80 (можно выбрать любой из незанятых).

```
server {
      listen 81;
      listen [::]:81;

      server_name example.ubuntu.com;

      root /var/www/tutorial;
      index index.html;

      location / {
            try_files $uri $uri/ =404;
      }
}
```

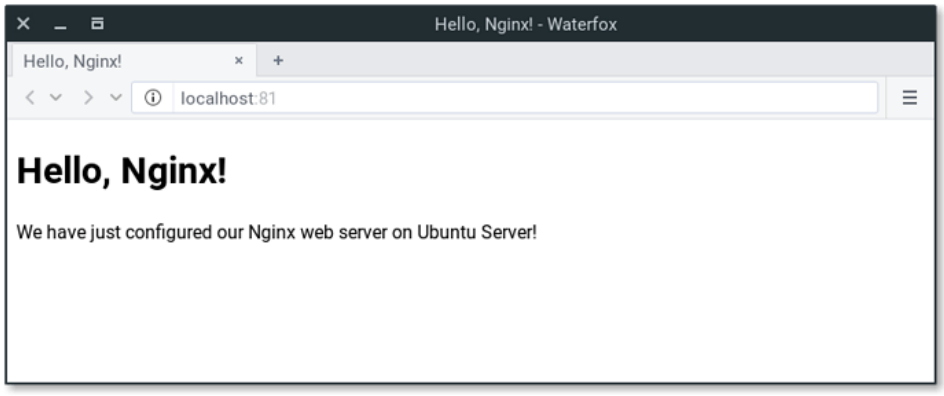
root – это директория, где хранится наш html-файл. **index** используется для определения файла, доступного в корневой директории сайта. **server_name** может быть любым, так как не указывает на реальный внешний домен.

Запуск виртуального хоста и тестирование

Для того, чтобы созданный сайт начал работать, достаточно перезапустить сервер **nginx**.

```
sudo service nginx restart
```

Для проверки, что сайт работает, можно в браузере открыть localhost с указанием выбранного порта или сделать запрос с помощью curl.



Docker

Docker – это программная платформа для быстрой разработки, тестирования и развертывания приложений. Docker упаковывает ПО в стандартизованные блоки, которые называются контейнерами. Каждый контейнер включает все необходимое для работы приложения: библиотеки, системные инструменты, код и среду исполнения. Благодаря Docker можно быстро развертывать и масштабировать приложения в любой среде и сохранять уверенность в том, что код будет работать.

В основе работы Docker лежит стандартизированный способ исполнения кода. Подобно тому как виртуальная машина создает виртуальное представление аппаратного обеспечения сервера (то есть устраняет необходимость непосредственно управлять таковым), контейнеры создают виртуальное представление серверной операционной системы. После установки на каждый сервер Docker предоставляет доступ к простым командам, необходимым для сборки, запуска или остановки контейнеров.

Docker помогает выкладывать код быстрее, быстрее тестировать, быстрее выкладывать приложения и уменьшить время между написанием кода и запуска кода. Docker делает это с помощью легковесной платформы контейнерной виртуализации.

В своем ядре Docker позволяет запускать практически любое приложение, безопасно изолированное в контейнере. Безопасная изоляция позволяет запускать на одном хосте много контейнеров одновременно.

Платформа и средства контейнерной виртуализации могут быть полезны в следующих случаях:

- упаковывание приложения (и так же используемых компонент) в docker-контейнеры;
- раздача и доставка этих контейнеров командам для разработки и тестирования;
- выкладывания этих контейнеров на продакшены, как в дата-центры, так и в облака.

Docker состоит из двух главных компонент:

- Docker: платформа виртуализации с открытым кодом;
- Docker Hub: платформа-как-сервис для распространения и управления docker контейнерами.

Чтобы понимать, из чего состоит docker, нужно знать о трех компонентах:

Образы

Docker-образ — это read-only шаблон. Например, образ может содержать операционку Ubuntu с Apache и приложением на ней. Образы используются для создания контейнеров. Docker позволяет легко создавать новые образы, обновлять существующие, или вы можете скачать образы созданные другими людьми. Образы — это компонента сборки docker-a.

Реестр

Docker-реестр хранит образы. Есть публичные и приватные реестры, из которых можно скачать либо загрузить образы. Публичный Docker-реестр — это Docker Hub. Там хранится огромная коллекция образов. Образы могут быть созданы или вы можете использовать образы созданные другими. Реестры — это компонента распространения.

Контейнеры

Контейнеры похожи на директории. В контейнерах содержится все, что нужно для работы приложения. Каждый контейнер создается из образа. Контейнеры могут быть созданы, запущены, остановлены, перенесены или удалены. Каждый контейнер изолирован и является безопасной платформой для приложения. Контейнеры — это компонента работы.

Каждый образ состоит из набора уровней. Docker использует union file system для сочетания этих уровней в один образ. Union file system позволяет файлам и директориями из разных файловых систем (разным ветвям) прозрачно накладываться, создавая когерентную файловую систему.

Одна из причин, по которой docker легковесен — это использование таких уровней. Когда вы изменяете образ, например, обновляете приложение, создается новый уровень. Так, без замены всего образа или его пересборки, как вам возможно придётся сделать с виртуальной машиной, только уровень добавляется или обновляется. И вам не нужно раздавать весь новый образ, раздается только обновление, что позволяет распространять образы проще и быстрее.

В основе каждого образа находится базовый образ. Например, ubuntu, базовый образ Ubuntu, или fedora, базовый образ дистрибутива Fedora. Так же вы можете использовать образы как базу для создания новых образов. Например, если у вас есть образ apache, вы можете использовать его как базовый образ для веб-приложений.

(!) Docker обычно берет образы из реестра Docker Hub.

Docker образы могут создаться из этих базовых образов, шаги описания для создания этих образов мы называем инструкциями. Каждая инструкция создает новый образ или уровень. Инструкциями будут следующие действия:

- запуск команды
- добавление файла или директории

- создание переменной окружения
- указания что запускать когда запускается контейнер этого образа

Эти инструкции хранятся в файле Dockerfile. Docker считывает это Dockerfile, когда вы собираете образ, выполняет эти инструкции, и возвращает конечный образ.

С помощью docker клиента вы можете искать уже опубликованные образы и скачивать их на вашу машину с docker для создания контейнеров.

Docker Hub предоставляет публичные и приватные хранилища образов. Поиск и скачивание образов из публичных хранилищ доступно для всех. Содержимое приватных хранилищ не попадает в результат поиска.

Контейнер состоит из операционной системы, пользовательских файлов и метаданных. Каждый контейнер создается из образа. Этот образ говорит docker-у, что находится в контейнере, какой процесс запустить, когда запускается контейнер и другие конфигурационные данные. Docker образ доступен только для чтения. Когда docker запускает контейнер, он создает уровень для чтения/записи сверху образа (используя union file system, как было указано раньше), в котором может быть запущено приложение.

Пример запуска контейнера:

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

Клиент запускается с помощью команды docker, с опцией run, которая говорит, что будет запущен новый контейнер. Минимальными требованиями для запуска контейнера являются следующие атрибуты:

- какой образ использовать для создания контейнера. В данном случае ubuntu
- команду которую вы хотите запустить когда контейнер будет запущен. В данном случае /bin/bash

Docker, по порядку, делает следующее:

- скачивает образ ubuntu: docker проверяет наличие образа ubuntu на локальной машине, и если его нет — то скачивает его с Docker Hub. Если же образ есть, то использует его для создания контейнера;
- создает контейнер: когда образ получен, docker использует его для создания контейнера;
- инициализирует файловую систему и монтирует read-only уровень: контейнер создан в файловой системе и read-only уровень добавлен образ;
- инициализирует сеть/мост: создает сетевой интерфейс, который позволяет docker-у общаться хост машиной;
- Установка IP адреса: находит и задает адрес;
- Запускает указанный процесс (приложение);
- Обрабатывает и выдает вывод приложения: подключается и логирует стандартный вход, вывод и поток ошибок приложения, что бы вы могли отслеживать как работает ваше приложение.

Docker использует технологию namespaces для организации изолированных рабочих пространств, которые мы называем контейнерами. Когда мы запускаем контейнер, docker создает набор пространств имен для данного контейнера.

Это создает изолированный уровень, каждый аспект контейнера запущен в своем простанстве имен, и не имеет доступ к внешней системе.

Docker использует технологию cgroups (Control groups, контрольные группы). Ключ к работе приложения в изоляции - предоставление приложению только тех ресурсов, которые вы хотите предоставить. Это гарантирует, что контейнеры будут хорошими соседями. Контрольные группы позволяют разделять доступные ресурсы железа и если необходимо, устанавливать пределы и ограничения. Например, ограничить возможное количество памяти контейнеру.

Установка Docker с помощью curl

```
# Скачиваем скрипт установки docker

curl -fsSL https://get.docker.com -o get-docker.sh

# запускаем скрипт

sudo sh get-docker.sh

# добавляем текущего пользователя в группу docker (чтобы использовать команды без sudo)

sudo usermod -aG docker $USER
```

где $USER – имя пользователя (например: sudo usermod -aG docker student).

В некоторых случаях этого может быть недостаточно для вызова Docker с правами суперпользователя (если слот Docker заблокирован), тогда необходимо дополнительно выполнить команду:

```
sudo chmod 666 /var/run/docker.sock
```

Пример запуска контейнера nginx

```
docker run --rm -p 8888:80 -v `pwd`:/app -it nginx
```

Опции:

--rm – удалить контейнер после остановки;

-p 8888:80 - связать локальный порт 8888 хостовой машины с портом 80 приложения в контейнере;

-d - контейнер в фоновом режиме, STDOUT приложения не выводится

-v `pwd`:/app - монтировать локальную папку (pwd - выводит путь к текущей директории) в папку /app внутри контейнера (для работы с внешними данными в контейнере);

-it - комбинация параметров i и t позволяет запустить контейнер в интерактивном режиме, таким образом, станет доступным консоль внутри контейнера.

Сборка и запуск нового образа

Для сборки нового образа на основе существующего необходимо создать файл **Dockerfile** и прописать в него несколько команд.

Создадим в директории файл **index.html** со следующим содержимым:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Docker Nginx</title>
</head>
<body>
  <h2>Hello from Nginx container</h2>
</body>
</html>
```

В той же директории создадим файл Dockerfile и пропишем в него следующие команды:

```
FROM nginx:latest

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

Построение модифицированного образа начинается с использования базового образа **nginx** (команда **FROM**). Если образ еще не использовался, он будет загружен на локальную машину и будет использоваться в построении модификации.

Команда **COPY** скопирует ранее созданный файл **index.html** в директорию **/usr/share/nginx/html** внутри контейнера, перезаписывая файл, который находится там по умолчанию (в образе **nginx:latest**).

Чтобы собрать образ, запустим следующую команду:

```
docker build -t webserver .
```

В случае, если сборка прошла успешно, мы можем запустить образ в контейнере:

```
docker run -it --rm -d -p 8080:80 --name web webserver
```

Можно убедиться, что nginx сервер запустился по адресу **http://localhost:8080** через браузер или запрос **culr**.

Другие команды Docker

Отобразить список запущенных контейнеров:

```
docker ps
```

Остановить работу контейнера:

```
docker stop [containerID]
```

Отобразить список доступных локально образов:

```
docker images
```

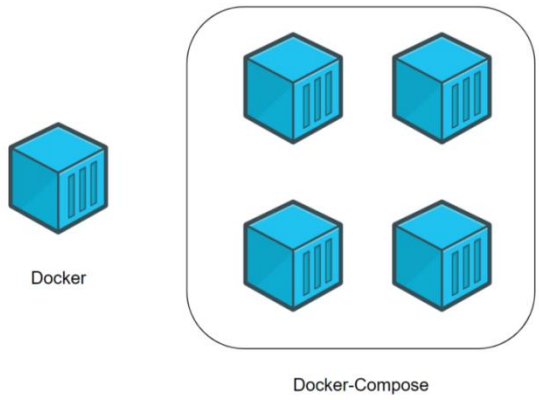
Docker Compose

Docker Compose — это инструментальное средство, входящее в состав Docker. Оно предназначено для решения задач, связанных с развёртыванием проектов. Реальные проекты обычно включают в себя набор совместно работающих приложений.

Если для обеспечения функционирования проекта используется несколько сервисов, то Docker Compose может пригодиться. Например, в ситуации, когда создают веб-сайт, которому, для выполнения аутентификации пользователей, нужно подключиться к базе данных. Подобный проект может состоять из двух сервисов — того, что обеспечивает работу сайта, и того, который отвечает за поддержку базы данных.

Docker применяется для управления отдельными контейнерами (сервисами), из которых состоит приложение.

Docker Compose используется для одновременного управления несколькими контейнерами, входящими в состав приложения. Этот инструмент предлагает те же возможности, что и Docker, но позволяет работать с более сложными приложениями.



Типичный сценарий использования Docker Compose

Представьте себе, что вы являетесь разработчиком некоего веб-проекта. В этот проект входит два веб-сайта. Первый позволяет людям, занимающимся бизнесом, создавать, всего в несколько щелчков мышью, интернет-магазины. Второй нацелен на поддержку клиентов. Эти два сайта взаимодействуют с одной и той же базой данных.

Ваш проект становится всё популярнее, и оказывается, что мощности сервера, на котором он работает, уже недостаточно. В результате вы решаете перевести весь проект на другую машину.

К сожалению, нечто вроде Docker Compose вы не использовали. Поэтому вам придётся переносить и перенастраивать сервисы по одному, надеясь на то, что вы, в процессе этой работы, ничего не забудете.

Если же вы используете Docker Compose, то перенос вашего проекта на новый сервер — это вопрос, который решается выполнением нескольких команд. Для того чтобы завершить перенос проекта на новое место, вам нужно лишь выполнить кое-какие настройки и загрузить на новый сервер резервную копию базы данных.

Разработка клиент-серверного приложения с использованием Docker Compose

В качестве примера разработаем небольшой веб-сайт (сервера) на Python, который умеет выдавать файл с фрагментом текста. Этот файл у сервера запрашивает программа (клиент), тоже написанная на Python. После получения файла с сервера программа выводит текст, хранящийся в нём, на экран.

Проверить, что Docker Compose установлен:

```
sudo apt install docker-compose
```

Для того чтобы построить клиент-серверное приложение, нужно создать папку проекта. Она должна содержать следующие файлы и папки:

- Файл **docker-compose.yml**. Это файл Docker Compose, который будет содержать инструкции, необходимые для запуска и настройки сервисов.
- Папка **server**. Она будет содержать файлы, необходимые для обеспечения работы сервера.
- Папка **client**. Здесь будут находиться файлы клиентского приложения.

В результате содержимое главной папки вашего проекта должно выглядеть так:

```
.
├── client/
├── docker-compose.yml
└── server/
2 directories, 1 file
```

Создание сервера

Перейдите в папку **server** и создайте в ней следующие файлы:

- Файл **server.py**. В нём будет находиться код сервера.
- Файл **index.html**. В этом файле будет находиться фрагмент текста, который должно вывести клиентское приложение.
- Файл **Dockerfile**. Это — файл Docker, который будет содержать инструкции, необходимые для создания окружения сервера.

Вот как должно выглядеть содержимое папки **server/**:

```
.
├── Dockerfile
```

```
├── index.html
└── server.py
0 directories, 3 files
```

Редактирование Python-файла.

Добавим в файл **server.py** следующий код:

```
#!/usr/bin/env python3

# Импорт системных библиотек python.
# Эти библиотеки будут использоваться для создания веб-сервера.
# Эти библиотеки обычно устанавливаются вместе с Python.
import http.server
import socketserver

# Эта переменная нужна для обработки запросов клиента к серверу.
handler = http.server.SimpleHTTPRequestHandler

# Сервер запустится на порте 1234.

with socketserver.TCPServer(("", 1234), handler) as httpd:
  # Благодаря этой команде сервер будет выполняться постоянно, ожидая
запросов от клиента.
  httpd.serve_forever()
```

Этот код позволяет создать простой веб-сервер. Он будет отдавать клиентам файл **index.html**, содержимое которого позже будет выводиться на веб-странице.

Редактирование HTML-файла

В файл **index.html** добавим следующий текст:

```
Docker-Compose is magic!
```

Этот текст будет передаваться клиенту.

Редактирование файла Dockerfile

Сейчас мы создадим простой файл **Dockerfile**, который будет отвечать за организацию среды выполнения для Python-сервера. В качестве основы создаваемого образа воспользуемся официальным образом, предназначенным для выполнения программ, написанных на Python. Вот содержимое Dockerfile:

```
# Dockerfile всегда должен начинаться с импорта базового образа.
# Для этого используется ключевое слово 'FROM'.
# Здесь нам нужно импортировать образ python (с DockerHub).
# В результате мы, в качестве имени образа, указываем 'python', а в качестве
версии - 'latest'.
FROM python:latest
```

```
# Для того чтобы запустить в контейнере код, написанный на Python, нужно импортировать файлы 'server.py' и 'index.html'.
# Для того чтобы это сделать, мы используем ключевое слово 'ADD'.
# Первый параметр, 'server.py', представляет собой имя файла, хранящегося на компьютере.
# Второй параметр, '/server/', это путь, по которому нужно разместить указанный файл в образе.
# Здесь мы помещаем файл в папку образа '/server/'.
ADD server.py /server/
ADD index.html /server/

# Здесь мы воспользуемся командой 'WORKDIR'.
# Она позволяет изменить рабочую директорию образа.
# В качестве такой директории, в которой будут выполняться все команды, устанавливаем '/server/'.
WORKDIR /server/
```

Создание клиента

Перейдите в папку вашего проекта **client** и создайте в ней следующие файлы:

- Файл **client.py**. Тут будет находиться код клиента.
- Файл **Dockerfile**. Этот файл играет ту же роль, что и аналогичный файл в папке сервера. А именно, он содержит инструкцию, описывающую создание среды для выполнения клиентского кода.

В результате папка **client/** на данном этапе работы должна выглядеть так:

```
.
├── client.py
└── Dockerfile
0 directories, 2 files
```

Редактирование Python-файла

Добавим в файл **client.py** следующий код:

```
#!/usr/bin/env python3

# Импортируем системную библиотеку Python.
# Она используется для загрузки файла 'index.html' с сервера.
# Эта библиотека устанавливается вместе с Python.

import urllib.request

# Эта переменная содержит запрос к 'http://localhost:1234/'.
# localhost указывает на то, что программа работает с локальным сервером.
# 1234 - это номер порта
fp = urllib.request.urlopen("http://localhost:1234/")
```

```
# 'encodedContent' соответствует закодированному ответу сервера ('index.html').
# 'decodedContent' соответствует раскодированному ответу сервера (тут будет то, что мы хотим вывести на экран).

encodedContent = fp.read()
decodedContent = encodedContent.decode("utf8")

# Выводим содержимое файла, полученного с сервера ('index.html').
print(decodedContent)

# Закрываем соединение с сервером.
fp.close()
```

Благодаря этому коду клиентское приложение может загрузить данные с сервера и вывести их на экран.

Редактирование файла Dockerfile

Как и в случае с сервером, создаём для клиента простой **Dockerfile**, ответственный за формирование среды, в которой будет работать клиентское Python-приложение. Вот код клиентского **Dockerfile**:

```
# То же самое, что и в серверном Dockerfile.
FROM python:latest

# Импортируем 'client.py' в папку '/client/'.
ADD client.py /client/

# Устанавливаем в качестве рабочей директории '/client/'.
WORKDIR /client/
```

Docker Compose

Мы создали два разных проекта: сервер и клиент. У каждого из них имеется собственный файл **Dockerfile**. Теперь же мы приступаем к работе с Docker Compose. Для этого обратимся к файлу **docker-compose.yml**, расположенному в корневой папке проекта.

Вот код, который нужно поместить в файл **docker-compose.yml**:

```
# Файл docker-compose должен начинаться с тега версии.
version: "3"

# Следует учитывать, что docker-compose работает с сервисами.
# 1 сервис = 1 контейнер.
# Сервисом может быть клиент, сервер, сервер баз данных...
# Раздел, в котором будут описаны сервисы, начинается с 'services'.
services:

  # Нам нужно два сервиса.
  # Первый сервис (контейнер): сервер.
  # Назвать его можно так, как нужно разработчику.
```

```
  # Понятное название сервиса помогает определить его роль.
  # Здесь мы, для именования соответствующего сервиса, используем ключевое
слово 'server'.
  server:
    # Ключевое слово "build" позволяет задать
    # путь к файлу Dockerfile, который нужно использовать для создания образа,
    # который позволит запустить сервис.
    # Здесь 'server/' соответствует пути к папке сервера,
    # которая содержит соответствующий Dockerfile.
    build: server/

    # Команда, которую нужно запустить после создания образа.
    # Следующая команда означает запуск "python ./server.py".
    command: python ./server.py

   # В нашем случае нужно использовать порт компьютера 1234 и организовать его
связь с портом 1234 контейнера (так как именно на этот порт сервер ожидает
поступления запросов).
    ports:
      - 1234:1234

  # Второй сервис (контейнер): клиент.
  # Этот сервис назван 'client'.
  client:

    # Здесь 'client/ соответствует пути к папке, которая содержит
    # файл Dockerfile для клиентской части системы.
    build: client/

    # Команда, которую нужно запустить после создания образа.
    # Следующая команда означает запуск "python ./client.py".
    command: python ./client.py

    # Ключевое слово 'network_mode' используется для описания типа сети.
    # Тут мы указываем то, что контейнер может обращаться к 'localhost'
компьютера.
    network_mode: host

    # Ключевое слово 'depends_on' позволяет указывать, должен ли сервис,
    # прежде чем запуститься, ждать, когда будут готовы к работе другие сервисы.
    # Нам нужно, чтобы сервис 'client' дождался готовности к работе сервиса
'server'.
    depends_on:
      - server
```

Сборка проекта

После того, как в **docker-compose.yml** внесены все необходимые инструкции, проект нужно собрать. Этот шаг нашей работы напоминает использование команды **docker build**, но соответствующая команда имеет отношение к нескольким сервисам:

```
docker-compose build
```

Запуск проекта

Теперь, когда проект собран, пришло время его запустить. Этот шаг работы соответствует шагу, на котором, при работе с отдельными контейнерами, выполняется команда **docker run**:

```
docker-compose up
```

После выполнения этой команды в терминале должен появиться текст, загруженный клиентом с сервера: **Docker-Compose is magic!**.

Как уже было сказано, сервер использует порт компьютера **1234** для обслуживания запросов клиента. Поэтому, если перейти в браузере по адресу http://localhost:1234/, в нём будет отображена страница с текстом **Docker-Compose is magic!**.

Полезные команды Docker-Compose

Останавка и удаление контейнеров и другие ресурсы, созданные командой **docker-compose up**:

```
docker-compose down
```

Вывод журналов сервисов:

```
docker-compose logs -f [service name]
```

С помощью такой команды можно вывести список контейнеров:

```
docker-compose ps
```

Данная команда позволяет выполнить команду в выполняющемся контейнере:

```
docker-compose exec [service name] [command]
```

Например, она может выглядеть так:

```
docker-compose exec server ls
```

Такая команда позволяет вывести список образов:

```
docker-compose images
```

Оркестрация контейнеров

Оркестрация или оркестровка — если обратиться к wiki — это автоматическое размещение, координация и управление сложными компьютерными системами и службами.

Технология контейнеризации в IT на сегодняшний день скорее правило, чем исключение. Контейнеры используются как в инфраструктуре, так и в процессах

разработки ПО. И когда в компании число контейнеров или микросервисов переваливает за много тысяч, появляется необходимость в системах оркестрации.

Типичный кейс: есть некая компания, которая занимается разработкой ПО и у нее имеется процесс CI/CD, например, на Docker + GitLab. Разработчики пишут код, упаковывают его в образы Docker, дальше сохраняют в локальном registry, на этом этап CI (Continuous Integration) обычно заканчивается.

Далее — этап CD (Continuous Deployment). Для него необходима некая среда запуска контейнеров для тестирования. Пока компания маленькая, число релизов и их частота небольшие, процесс работает. Число виртуальных машин растет, и их приводят в порядок системами оркестрации контейнеров.

Основные игроки на рынке

Задачи оркестрации решают такие продукты, как Kubernetes, Docker Swarm, Apache Mesos — это не единственные продукты на рынке. Также есть системы оркестрации контейнеров Nomad, Amazon EC2 Container Service, Microsoft Azure Container Service, однако они менее популярны. Каждый продукт имеет свои особенности. По большому счету, они все решают две основные задачи:

- Динамическое распределение контейнеров по узлам кластера
- Стандарт описания приложения, например, yaml, docker compose и другие

Система оркестрации Kubernetes

Универсальный солдат, появился в результате наработок Google. В 2014 г. Google открыла код Kubernetes и стала распространять систему под лицензией Apache 2.0. В результате началось быстрое развитие системы сообществом. На данный момент — это стабильный продукт, который одинаково хорошо чувствует себя как в небольших проектах, так и Enterprise сегменте.

Система орекстрации Docker Swarm

Компания Docker одна из первых предложила реализацию контейнеров и управление ими. Собственно Kubernetes внутри себя обычно использует Docker как среду исполнения контейнеров (container runtimes). Docker swarm позволяет объединять Docker хосты в общий виртуальный хост. Обычно его используют в небольших проектах. Продукт на данный момент активно развивается.

Apache Mesos

Apache Mesos — это централизованная отказоустойчивая система для управления кластером. Объединяет в группы отдельные узлы, согласно требованиям, а также обеспечивает их изоляцию от остальных IT-ресурсов.

Это разработка университета Беркли, как продукт вышла на рынок в 2009 г. Apache Mesos имеет нестандартный подход к своей архитектуре. Он объединяет существующие объекты в единый виртуальный ресурс, формируя крупные кластеры и эффективную систему управления серверной инфраструктурой, где каждому кластеру выделяется свой

индивидуальный пул ресурсов. Хорошо подходит для очень больших систем и масштабных проектов.

Оркестрации на примере Kubernetes:

- С помощью своего планировщика (kube-scheduler) в режиме реального времени распределяет контейнеры по рабочим нодам. Если ресурсы ноды заканчиваются — выполняет миграцию на доступные по ресурсам узлы.
- Позволяет запускать сразу несколько реплик приложения и следить за их доступностью. Тем самым обеспечивает отказоустойчивость приложения внутри кластера.
- Стандартизирует инфраструктуру (Infrastructure as a Code).
- Обеспечивает удобный мониторинг.

В **Kubernetes** действительно много хорошего функционала для администрирования контейнеров, с которым стоит познакомиться.

Jenkins

Jenkins – программная система с открытым исходным кодом на Java, предназначенная для обеспечения процесса непрерывной интеграции программного обеспечения. Практика CI/CD широко распространена в современном мире у крупных компаний, где практикуются ежедневные изменения тестовых и релизных версий продуктов.

Пример разворачивания Jenkins в контейнере для домашнего тестирования:

https://habr.com/ru/articles/677142/

Задание на лабораторную работу

Цель: Научиться устанавливать, настраивать **Docker** и подготавливать к работе контейнеры. Освоить работу с **Docker-compose**.

1. Установить **curl**.
2. Проверить, что **curl** возвращает содержимое html-страницы произвольного сайта.
3. Установить **nginx** (или использовать ранее установленный **nginx** из прошлой лабораторной), запустить, убедиться, что содержимое тестовой страницы отображается в браузере, доступно по запросу **curl**.
4. Заменить страницу, которую отдает **nginx** по умолчанию или добавить новую на отдельный виртуальный хост в соответствии с примером в методическом указании.
5. Установить **Docker** с помощью **curl**, настроить его и проверить работу с помощью тестового образа.
6. Запустить **nginx** в контейнере, проверить его работу в соответствии с примером в методическом указании (через веб-браузер и с помощью **curl**).

7. Собрать модификацию образа **nginx**, заменив в нем страницу, отдаваемую по умолчанию, запустить контейнер с этим образом и проверить его работу через браузер или **curl**-запрос.

8. Создать новый проект **docker-compose** в соответствии с примером в методическом указании, собрать проект, убедиться, что сборка проходит корректно.

9. Запустить проект **docker-compose**, убедиться, что в результате клиент получает от сервера html-файл. Проверить, что сервер также доступен через браузер.

10. Остановить сервер, заменить содержимое html-страницы на свое, собрать и запустить проект заново, убедиться, что по клиенту доступна новая страница.

Результат: Установили, настроили **Docker** и развернули контейнер с приложением. Изучили **Docker Compose** и развернули среду с двумя контейнерами с его помощью.

ТРЕБОВАНИЯ К ОТЧЕТУ

Отчет должен содержать следующие разделы:

1. Титульный лист, оформленный согласно утвержденному образцу.
2. Цели выполняемой лабораторной работы.
3. Задание на лабораторную работу.
4. Описание процесса выполнения работы: для каждого действия, производимого в командной строке, в отчет следует включить:
   - краткое описание действия;
   - вводимая команда или команды;
   - реакция системы на ввод команд (если объем выводимых данных превышает несколько строк, всю информацию включать в отчет не следует).
5. Выводы.