Системная и программная инженерия

1. Знакомство с Docker

**Введение**

Прежде чем рассказывать про Docker, нужно сказать несколько слов о технологии контейнеризации.

* ***Контейнеры*** — это способ упаковать приложение и все его зависимости в единый образ. Этот образ запускается в изолированной среде, не влияющей на основную операционную систему. Контейнеры позволяют отделить приложение от инфраструктуры: разработчикам не нужно задумываться, в каком окружении будет работать их приложение, будут ли там нужные настройки и зависимости. Они просто создают приложение, упаковывают все зависимости и настройки в единый образ. Затем этот образ можно запускать на других системах, не беспокоясь, что приложение не запустится.
* ***Docker*** — это платформа для разработки, доставки и запуска контейнерных приложений. Docker позволяет создавать контейнеры, автоматизировать их запуск и развертывание, управляет жизненным циклом. Он позволяет запускать множество контейнеров на одной хост-машине.

Контейнеризация похоже на виртуализацию, но это не одно и то же. Виртуализация работает как отдельный компьютер, со своим виртуальным оборудованием и операционной системой. При этом внутри одной ОС можно запустить другую ОС. В случае контейнеризации виртуальная среда запускается прямо из ядра основной операционной системы и не виртуализирует оборудование. Это означает, что контейнер может работать только в той же ОС, что и основная. При этом так как контейнеры не виртуализируют оборудование, они потребляют намного меньше ресурсов.

**Преимущество использования Docker:**

* Изоляция запущенных приложений. Можно отделить ваше приложение от ОС хоста и от других приложений;
* Облегчение процесса разработки, тестирования и развертывания приложений;
* Нет необходимости в конфигурации среды для запуска – она поставляется вместе с приложением.

**Компоненты Docker**

Выделим важные компоненты docker:

* **Docker image (докер-образ)** - это неизменяемый образ, из которого разворачивается контейнер. Его можно рассматривать как набор файлов, необходимых для запуска и работы приложения на другом хосте. Говоря простыми словами, образ Docker это образ, содержащий внутри себя приложение и все зависимости, необходимые для запуска этого приложения в Docker;
* **Docker registry** - это репозиторий, в котором хранятся образы. Пример: [DockerHub](https://hub.docker.com/),
* **Dockerfile (докер-файл)** - это файл-инструкция для сборки образа;
* **Docker container (докер-контейнер)** - это запущенный экземпляр образа Docker;
* **Docker host (докер-хост)** - машина с установленным Docker;
* **Docker daemon (докер-демон)** - это фоновый процесс, который работает постоянно и ожидает команды. Все операции по управлению контейнерами отправляются именно в демон, например: запустить или остановить контейнер, скачать образ. И уже на основе этих команд демон выполняет необходимые действия с контейнерами и образами. Поэтому докер-демон знает все о контейнерах, запущенных на одном хосте: сколько всего контейнеров, какие из них работают, где хранятся образы и так далее;
* **Docker client (докер-клиент)** - это клиент, при помощи которого пользователи взаимодействуют с демоном и отправляют ему команды.

**Практическая часть**

Для начала запустим первый ознакомительный контейнер [hello-world](https://hub.docker.com/_/hello-world), для этого воспользуемся следующей командой:

**docker run [container name]**

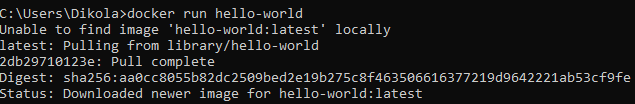


Рис. 1. Первичный запуск контейнера

Как мы видим, в ходе запуска контейнера Docker не нашел образ, на основе которого необходимо создать контейнер, поэтому он предварительно обращается к Docker-registry для того, чтобы «спулить» (клонировать) этот образ на локальную машину. При последующих запусках контейнеров, которые содержат в себе этот образ Docker не будет обращаться к Docker-registry, а будет использовать уже склонированный. После успешного клонирования Docker создает контейнер, и запускает его.

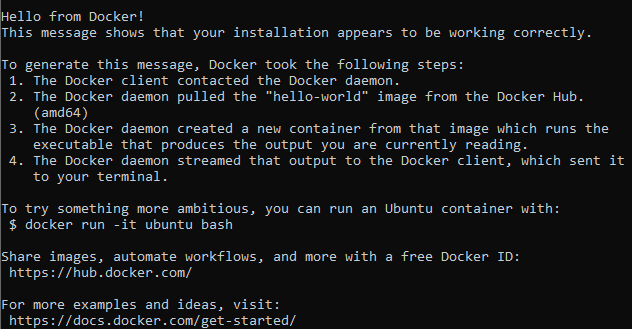


Рис. 2. Запущенный контейнер «hello-world»

Для просмотра списка активных контейнеров используется команда:

**docker ps**



Рис. 3. Выполнение команды «docker ps»

Мы видим, что контейнеров нету, дело в том, что текущая команда отображает только запущенные контейнеры. Для просмотра списка всех контейнеров используется команда:

**docker ps -a**



Рис. 4. Выполнение команды «docker ps -a»

Из полученного ответа видно, что статус контейнера ***Exited 0***. Контейнер запустил внутри себя команду, программа выполнила работу, и контейнер завершил свою работу. Это стандартная работа контейнера. Контейнер находится в активном состоянии только при условии, что запущенное в нем приложение работает.

Также мы можем просмотреть список образов командой:

**docker images**



Рис. 5. Выполнение команды «docker images»

**Работа внутри контейнера**

Прежде чем приступить к работе в контейнере предварительно скачаем образ [Busybox](https://hub.docker.com/_/busybox). Команда pull скачивает образ busybox из регистра Докера, и сохраняет его локально. Для этого выполним команду:

**docker pull [image-name]**

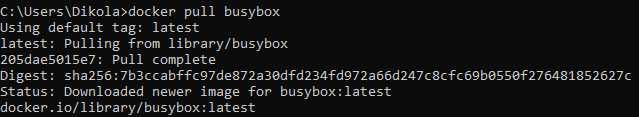


Рис. 6. Выполнение команды «docker pull»

Далее запустим контейнер, и зайдем внутрь него при помощи команды:

**docker run -it [image-name] sh**

Параметр -it используется для того, чтобы зайти внутрь контейнера, в качестве оболочки взаимодействия мы выбрали sh.



Рис. 7. Выполнение команды «docker run» для работы с контейнером изнутри

Внутри данного контейнера мы можем работать с ним, как с UNIX системой, и выполнять системные команды.

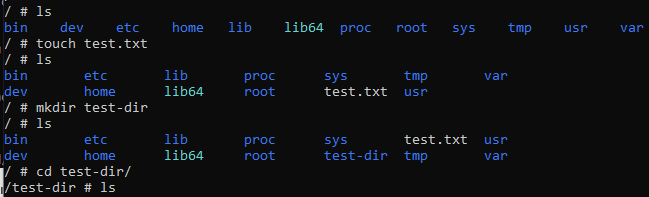


Рис. 8. Пример работы с контейнером изнутри

Для того, чтобы выйти из контейнера необходимо выполнить команду:

**exit**



Рис. 9. Выполнение команды «exit»

Если мы снова выполним команду **docker run -it busybox sh**, то, вместо того, чтобы вернуться в уже созданный контейнер, мы создадим новый на основе скачанного образа, и наши изменения, которые мы сделали в прошлом контейнере не будут совершены в новом. Для того, чтобы вернуться в уже созданный нами контейнер, необходимо при помощи команды **docker ps -a** узнать его id.

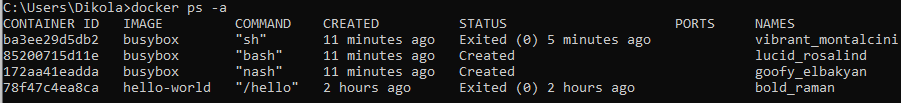


Рис. 10. Просмотр списка контейнеров

Как мы видим, id нашего контейнера **ba3ee29d5db2**. Далее нам необходимо запустить контейнер командой:

**docker start [container\_id]**

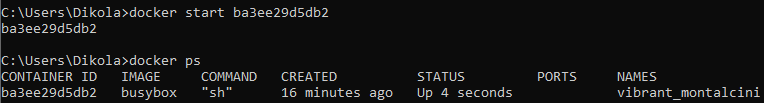


Рис. 11. Выполнение команды «docker start»

Для подключения к контейнеру выполните команду:

**docker attach [container\_id]**

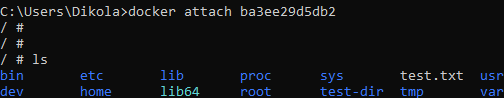


Рис. 12. Выполнение команды «docker attach»

Для остановки работы контейнера выполните команду:

**docker stop [container\_id]**

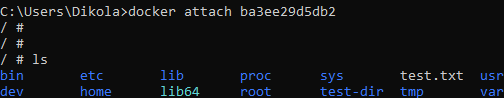


Рис. 13. Выполнение команды «docker stop»

Для удаления контейнера выполните команду:

**docker rm [container\_id]**



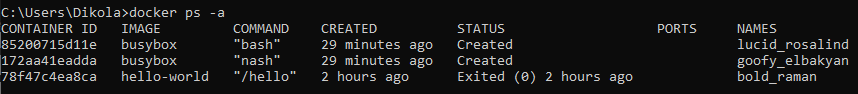


Рис. 14. Выполнение команды «docker rm»

Для удаления образа выполните команду:

**docker rmi [image\_id]**

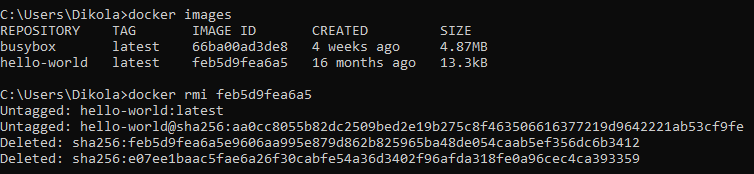


Рис. 15. Выполнение команды «docker rmi»

**Docker-compose**

Для упрощения работы с docker существуют много различных инструментов. Одним из популярных является docker-compose. Docker-compose - это приложение, написанное на Python, предназначенное для решения задач, связанных с развёртыванием проектов. Он использует заранее подготовленный **docker-compose.yml** с описанными инструкциями по запуску контейнеров (контейнеров может быть несколько). Перейдем в пустую директорию и создадим в ней **docker-compose.yml** файл со следующим содержимым:

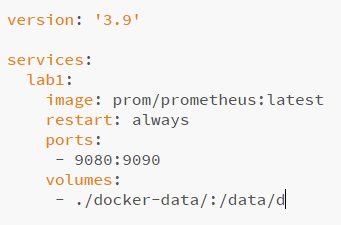


Рис. 16. Содержимое файла «docker-compose.yml»

Что указано в данном файле?

* version - это версия приложения docker-compose;
* services – это заголовок, в котором указывается перечисление контейнеров для запуск (на текущий момент приложение одно, приложение prometheus);
* image – образ, который будет использован для запуска контейнера;
* restart - политика перезапуска контейнера (always означает, что если в приложении возникнет ошибка, контейнер перезапустится);
* volumes - указывает на массив монтированных директорий, файлов. Так как в примере запускается БД, необходимо сохранять данные во вне контейнера. **./docker-data/:/data/db** указывает на то, что необходимо монтировать папку **./docker-data/**на локальной машине в папку **/data/db** внутри контейнера. Таким образом можно передавать различные файлы внутрь программы (например, конфиги приложения), а также сохранять полученные файлы во время работы контейнера (данные БД);
* ports - список портов, которые нам необходимо прокинуть из контейнера во вне. В Docker по умолчанию имеется своя подсеть. При создании контейнера, каждому из них выдается свой адрес в этой подсети (взаимодействие идет через NAT). Контейнер может взаимодействовать с внешней сетью через общий шлюз докер подсети (например, скачивать внутрь себя приложения). Но из вне (за NAT-ом) мы не может взаимодействовать с внутренними приложениями контейнера (обращаться к приложению внутри докера по определенному порту). Для этого используется проброс портов через NAT, который указывается через переменную ports. Таким образом, обращаясь к адресу **0.0.0.0:9080**из локальной консоли вы будете обращаться к приложению, работающему по адресу **0.0.0.0:9090** внутри контейнера.

Затем запустим контейнер при помощи команды (для выполнения необходимо находиться в директории с конфигурационным файлом):

**docker compose up -d**

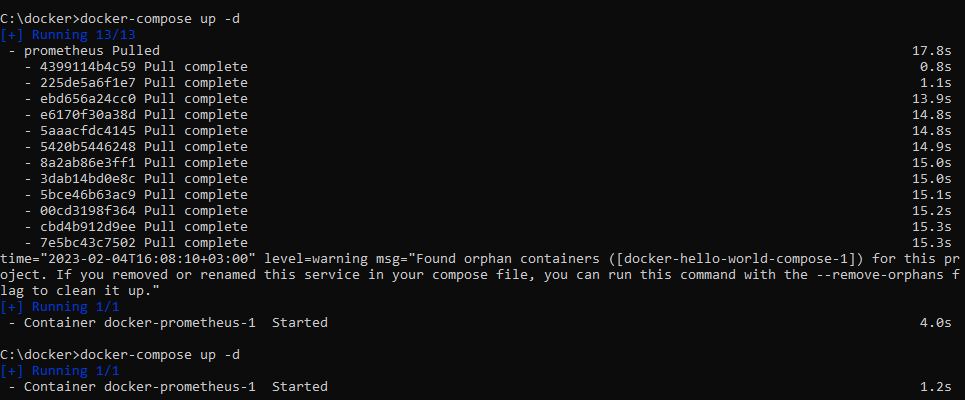


Рис. 17. Выполнение команды «docker-compose up -d»

После запуска контейнера перейдем в браузер по адресу <http://localhost:9080>, и мы увидим интерфейс приложения, запущенного в контейнере.

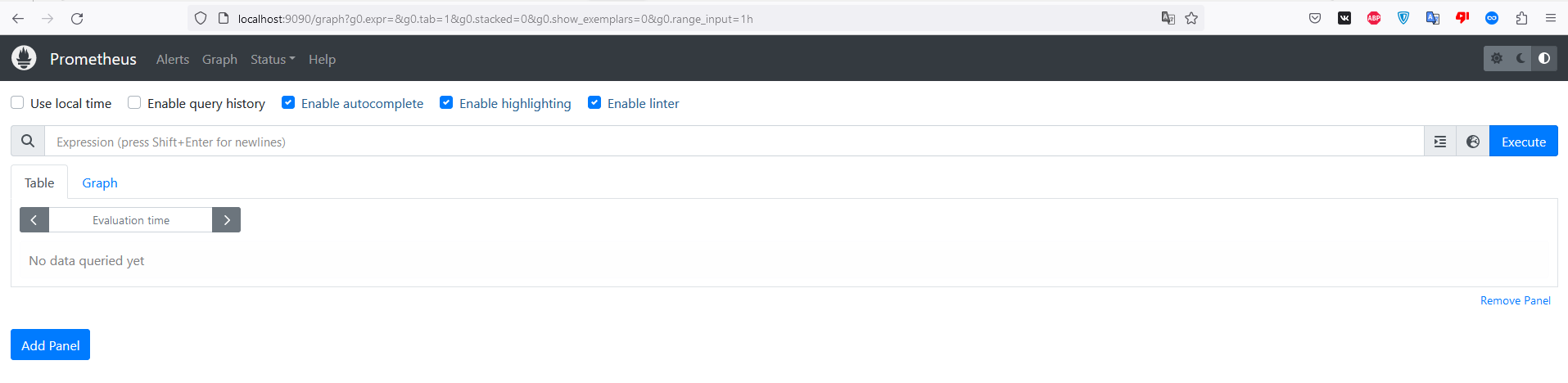


Рис. 18. Демонстрация работы приложения, запущенного в docker контейнере

Для того, чтобы подключиться во внутрь контейнера необходимо выполнить команду:

**docker exec -it [container\_id] sh**

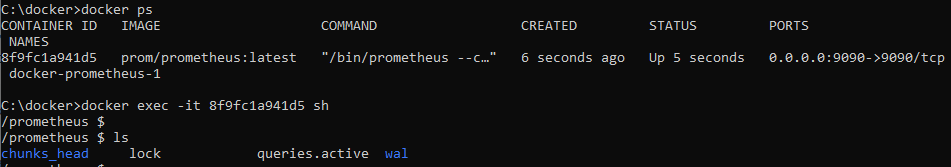


Рис. 19. Выполнение команды «docker exec»

Для остановки контейнера необходимо выполнить команду:

**docker-compose down**



Рис. 20. Выполнение команды «docker-compose down»

**Практическое задание**

1. Создайте контейнер, который будет использовать образ «busybox». Зайдите внутрь контейнера, и создайте текстовый файл по шаблону «ФИО\_группа».
2. При помощи docker-compose.yml создайте конфигурационный файл с любым образом, и запустите его. Продемонстрируйте работу приложения, которое вы запустили. Название сервиса в конфигурационном файле должно быть выполнено по шаблону по шаблону «ФИО\_группа».

**Теоретические вопросы**

1. **В чем отличие контейнера от образа?**

Контейнер – это способ упаковать приложение и все его зависимости в единый образ. Образ – набор файлов, необходимых для запуска и работы приложения на другом хосте.

1. **Какими способами можно создать контейнер?**

docker run [container name] – пулим образ и докер сам создает контейнер.

DockerFile – набор инструкций для сборки образа

1. **Какой командой можно просмотреть список активных контейнеров?**

docker ps -a

1. **Какой командой можно просмотреть список образов?**

docker images

1. **Откуда будет взят образ для контейнера, если его нет на локальной машине?**

В репозиторий docker контейнеров/изображений (По дефолту docker hub)

1. **Какой командой можно удалить контейнер?**

docker rm [container\_id]

1. **Что будет, если попытаться удалить образ, который используется другими контейнерами?**
2. **Для чего применяется файл docker-compose.yml?**

docker-compose.yml - инструкции по запуску контейнеров (контейнеров может быть несколько)

1. **Каким образом можно попасть вовнутрь контейнера?**

docker attach [container\_id]

1. **Для чего применяются volumes?**

volumes - указывает на массив монтированных директорий, файлов. Так как в примере запускается БД, необходимо сохранять данные во вне контейнера. **./docker-data/:/data/db** указывает на то, что необходимо монтировать папку **./docker-data/**на локальной машине в папку **/data/db** внутри контейнера. Таким образом можно передавать различные файлы внутрь программы (например, конфиги приложения), а также сохранять полученные файлы во время работы контейнера (данные БД);

1. **Каким образом работает проброс портов в Docker?**

В Docker по умолчанию имеется своя подсеть. При создании контейнера, каждому из них выдается свой адрес в этой подсети (взаимодействие идет через NAT). Контейнер может взаимодействовать с внешней сетью через общий шлюз докер подсети (например, скачивать внутрь себя приложения). Но из вне (за NAT-ом) мы не может взаимодействовать с внутренними приложениями контейнера (обращаться к приложению внутри докера по определенному порту). Для этого используется проброс портов через NAT, который указывается через переменную ports. Таким образом, обращаясь к адресу **0.0.0.0:9080**из локальной консоли вы будете обращаться к приложению, работающему по адресу **0.0.0.0:9090** внутри контейнера.