# **Docker**

**Краткая справка**

Технологии контейнеризации приложений нашли широкое применение в сферах разработки ПО и анализа данных. Эти технологии помогают сделать приложения более безопасными, облегчают их развёртывание и улучшают возможности по их масштабированию. Рост и развитие технологий контейнеризации можно считать одним из важнейших трендов современности.

Docker — это система управления контейнерами. Контейнеры же представляют собой логическое эволюционное продолжение виртуальных машин. Это изолированная среда для разработки и тестирования программного обеспечения.

С другими словами Docker — это платформа, которая предназначена для разработки, развёртывания и запуска приложений в контейнерах. Слово «Docker» в последнее время стало чем-то вроде синонима слова «контейнеризация». И если вы ещё не пользуетесь Docker, но при этом работаете или собираетесь работать в сферах разработки приложений или анализа данных, то Docker — это то, с чем вы непременно встретитесь в будущем.

Контейнер Docker потребляет мало ресурсов и быстро запускается, а еще его легко переносить с одного устройство на другое. Из-за этих преимуществ Docker постепенно наращивает аудиторию и превращается в некий индустриальный стандарт, которым пользуются даже крупные корпорации вроде Microsoft.

**Немного истории[2]**

Чтобы понять суть докера, сперва углубимся в историю. Раньше доставка грузов по всему миру осуществлялась в бочках, коробках и прочих ёмкостях. Проблема была в том, что все они были разных размеров и форм. Для разгрузки корабля требовалась уйма времени, а время увеличивало стоимость доставки.



Никто не мог ускорить процесс, ведь для разгрузки требовались люди и добавляя больше людей увеличивалась стоимость, добавляя их меньше – увеличивалось время.

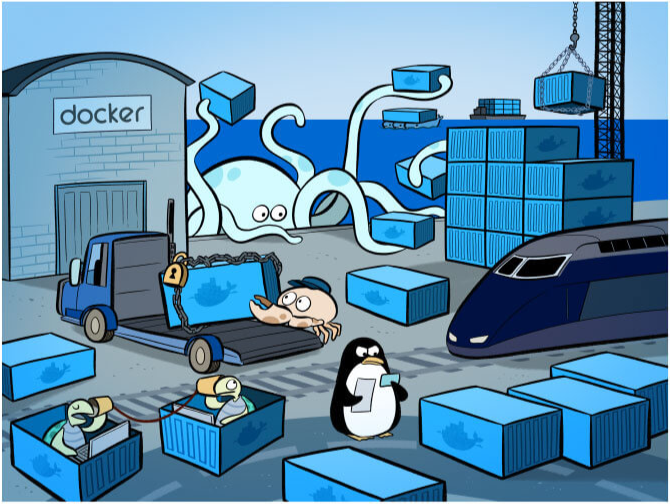
Все было достаточно печально до тех пор, пока не изобрели контейнеры. Те самые контейнеры, что сегодня можно встретить на любом корабле.



Они стали феноменом! Теперь не важно что конкретно находится внутри самой ёмкости, так как сама ёмкость одной формы. Систематизировав эту, казалось бы, мелочь, люди смогли построить по всему миру одинаковые краны для разгрузки. Тем самым сроки разгрузки и погрузки сократились до минимума.

**При чем здесь Докер?**

Но при чем здесь Докер? Докер сделал точно то же действие, но в программировании. Ранее для открытия проекта требовалось установить много разных технологий и настроить все на устройстве. Вам нужно открыть веб сайт, написанный на PHP? В таком случае установите локальный сервер, настройте Apache, запустите весь проект и о чудо – не получите при этом никаких ошибок. И вроде просто, но что если вам пришлют какой-то проект с технологией и языком, с которым вы до этого никогда не работали? У вас уйдет много времени чтобы все это протестировать.



Докер кардинально изменил подход, предложив систему контейнеров. Представим что вы создали проект и сам проект может быть написан на чем-угодно: Питон, Джава, С++ или любой другой язык. Вы запихиваете ваш проект в контейнер, а вместе к проекту добавляете всё необходимое окружение: среду выполнения, сервер, различные пакеты, приложения и всё остальное. Далее при помощи специальных команд вы можете выгрузить ваш проект (контейнер) и после передать его клиенту, начальнику или другому разработчику.

Если вы писали проект на Питон, а отправили его Джава разработчику, то ему не составит труда запустить среду. Процесс запуска всех проектов один и тот же, вне зависимости от того, что находится внутри. Вы можете создать проект, поместить туда что-угодно, а для запуска всегда будет использоваться одна и та же команда, при чем не важно на каком ПК вы будете запускать проект.

**Где применяется Докер?**

Докер применяется повсеместно. Если вы делаете проект для себя, на работе или делаете проект для заказчика – вы повсюду можете использовать Докер.



Если вы отправляете проект без докера, то вам еще потребуется описать процесс настройки проекта и процесс его запуска. При этом никто не исключает, что клиент может ошибиться и у него будет ошибка в ходе тестирования. В таком случае нужно будет уделить время и все показать самостоятельно. Если же воспользоваться Докером, то процесс тестирования свидеться к нескольким вещам: человек должен установить у себя программу Докер и далее через терминал при помощи одной команды сможет запустить ваш проект. При этом нет опаски что что-то пойдет не так. Вся среда будет в самом контейнере, поэтому ничего не потребуется устанавливать или настраивать.

**Основные команды Докер**

Управлять Докером можно при помощи специальной программы – Docker Desktop. Однако такой способ работы с Докер не всегда будет удобный и не всегда вам подойдет. За урок мы научимся прописывать Docker команды через терминал. Мы научимся отслеживать информацию про Докер и даже выполним запуск нескольких проектов.

Технология Докер в основном используется внутри проектов. Для создания и выполнения каких-либо действий разработчику приходится прописывать команды в терминале. По этой причине вам не столь важно уметь работать с программой Docker Desktop, как уметь прописывать верные команды в терминале. С некоторыми основными командами мы как раз и ознакомимся.

1. Установка и настройка

<https://medium.com/devops-with-valentine/how-to-install-docker-on-windows-10-11-step-by-step-83074a80e6f9>

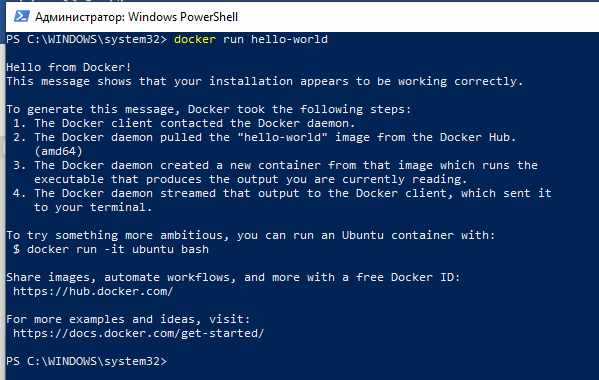
**Проверьте правильность установки**

Откройте терминал командной строки и запустите несколько команд Docker, чтобы убедиться, что Docker работает должным образом.

Некоторые хорошие команды, которые стоит попробовать, — docker versionэто проверить, установлена ​​ли у вас последняя версия, и docker psпосмотреть, есть ли у вас запущенные контейнеры. (Возможно, нет, поскольку вы только начали.)

Введите docker run hello-world команду и нажмите RETURN.

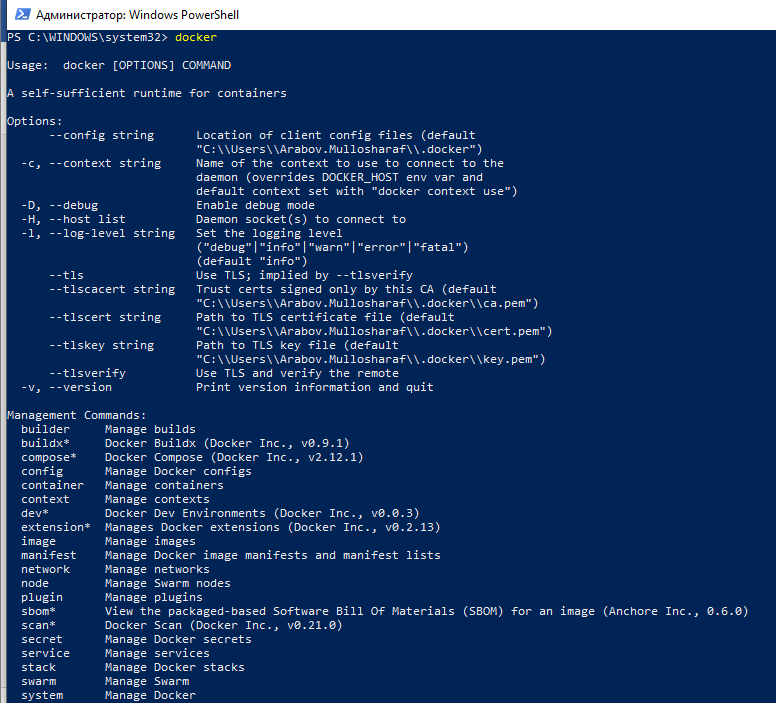
Команда выполняет некоторую работу за вас, если все работает хорошо, вывод команды выглядит следующим образом:



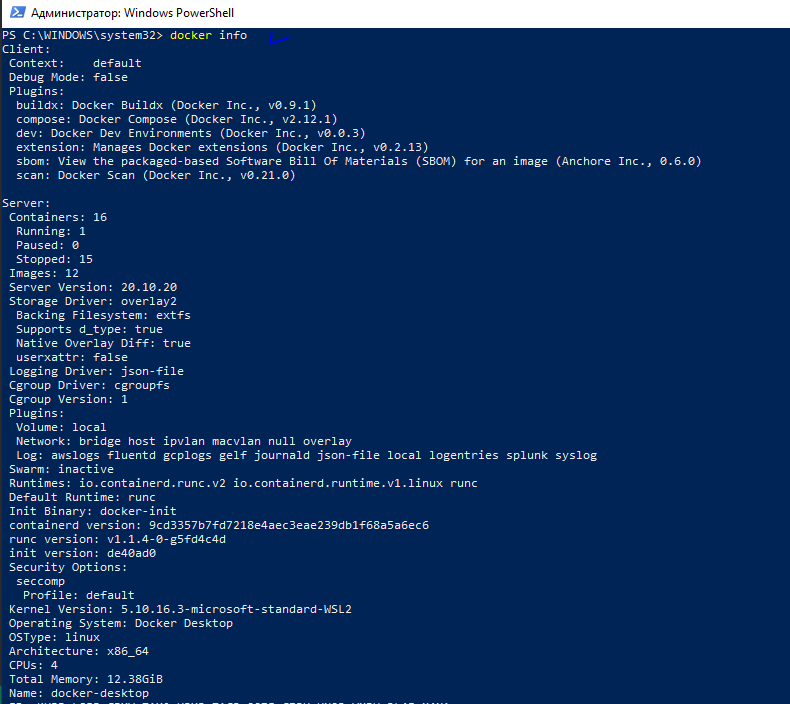
**Информационные команды**

Это команды, что отвечают за выдачу информацию относительно Докера, запущенных в ней контейнеров и образов. Наиболее простой командой является команда docker. При её выполнении вы получаете список всех возможных команд вместе с их кратким описанием.

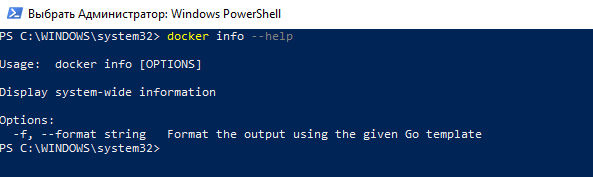
Если мы откроем Power Shell на имя администратора и напишем команду docker, то получим список необходимых команд DOKER:



Для выяснения состояния Докера: какие проекты запущены, что сейчас остановлено, а что на паузе, – для всего этого вы можете использовать команду docker info.

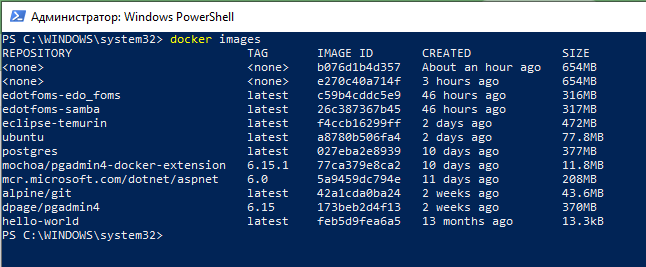


Кстати, если требуется узнать информацию по поводу команды или вспомнить что она делает, то всегда можно прописать опцию --help в конце: docker info --help.

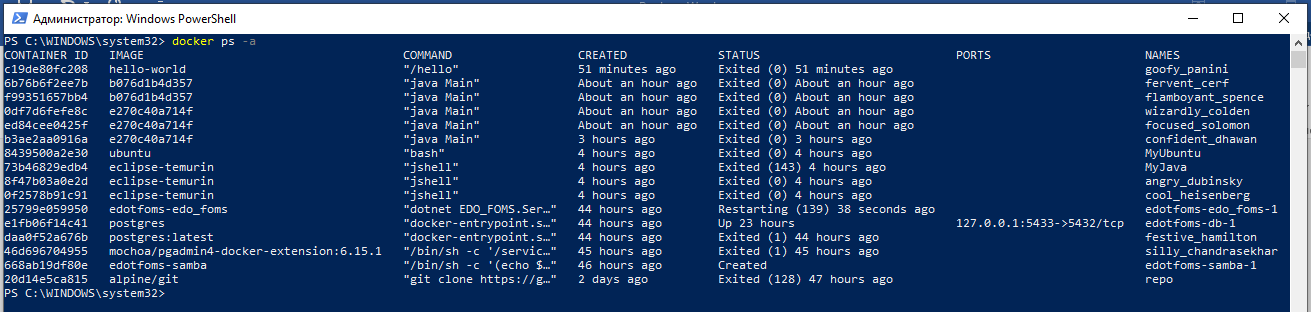


**Контейнеры и образы**

Посмотреть информацию про все запущенные образы можно при помощи команды docker images. Эта команда отображает список всех образов в вашем проекте.



Для отображения информации относительно всех контейнеров можно использовать команду docker ps -a.



**Контейнеры и образы в Docker**

Контейнеры и образы – основа любого Docker проекта. В ходе урока мы рассмотрим отличия между этими вещами. Дополнительно мы установим несколько образов и создадим пару проектов с использованием разных языков и технологий.

**Метафоры и Docker**

Мы постоянно сталкиваемся с метафорами. Если заглянуть в словарь Ожегова, то окажется, что метафора — это «скрытое образное сравнение, уподобление одного предмета, явления другому». Метафоры помогают нам ухватывать суть новых для нас явлений. Например, виртуальные контейнеры можно сравнить с обычными пластиковыми контейнерами. Такое сравнение, через сопоставление уже известных нам свойств обычных контейнеров со свойствами виртуальных контейнеров, поможет сначала с ними познакомиться, а потом и понять их сущность.



Как и обычный пластиковый контейнер, контейнер Docker обладает следующими характеристиками:

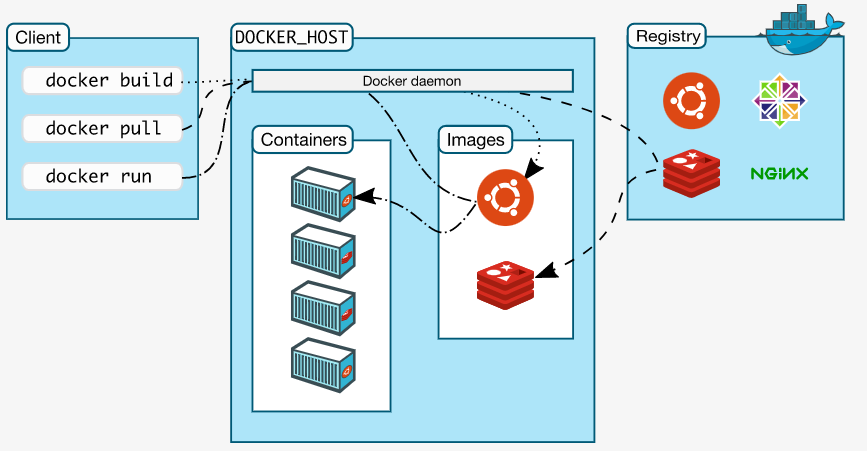
1. В нём можно что-то хранить. Нечто может находиться либо в контейнере, либо за его пределами.
2. Его можно переносить. Контейнер Docker можно использовать на локальном компьютере, на компьютере коллеги, на сервере поставщика облачных услуг (вроде AWS). Это роднит контейнеры Docker с обычными контейнерами, в которых, например, перевозят разные милые сердцу безделушки при переезде в новый дом.
3. В контейнер удобно что-то класть и удобно что-то из него вынимать. У обычного контейнера есть крышка на защёлках, которую надо снять для того, чтобы что-то положить в контейнер или что-то из него вынуть. У контейнеров Docker есть нечто подобное, представляющее их интерфейс, то есть — механизмы, позволяющие им взаимодействовать с внешним миром. Например, у контейнера есть порты, которые можно открывать для того, чтобы к приложению, работающему в контейнере, можно было бы обращаться из браузера. Работать с контейнером можно и средствами командной строки.
4. Если вам нужен контейнер, его можно заказать в интернет-магазине. Пустой контейнер можно купить, например, на сайте Amazon. В этот магазин контейнеры попадают от производителей, которые делают их в огромных количествах, используя пресс-формы. В случае с контейнерами Docker то, что можно сравнить с пресс-формой, а именно — образ контейнера, хранится в специальном репозитории. Если вам нужен некий контейнер, вы можете загрузить из репозитория соответствующий образ, и, используя его, этот контейнер создать.

Конечно, пластиковые контейнеры, в отличие от контейнеров Docker, никто вам не будет присылать бесплатно, да и когда вы их получите, они будут пустыми. А вот в контейнерах Docker всегда есть что-то интересное.

Docker работает по достаточно простому для понимания принципу. В любой проект вы можете добавить какой-либо язык программирования: Джава, Питон, PHP; можете добавить технологии: локальный сервер, база данных; и можете вовсе добавить полноценную операционную систему по типу Ubuntu.

Для добавления новых возможностей в проект, вам стоит скачивать образы (Image). Их может быть несколько и за счет них вы наполняете общий контейнер (проект). Контейнер состоит из образов (готовых решений) и вашего кода. По итогу вы получаете готовую закрытую среду, где сразу есть все технологии, код и даже операционная система.

Пример работы Docker представлен на фото ниже:



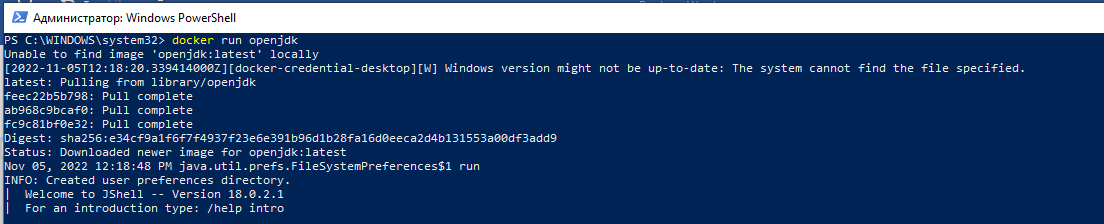
На фото также можно увидеть регистр. Он представляет из себя место, из которого Докер выбирает все образы и помещает их внутрь ваших проектов. Одним из таких регистров является Docker Hub. C ним вам придется постоянно работать, так как он содержит описание всех образов.

Управление контейнерами и образами в Docker мало чем отличается от управления другими приложениями в терминале Linux. Принцип тот же – вводим текст и получаем результат. Сначала надо указать ключевое слово Docker, потом указать команду, которую надо выполнить, а затем объект применения команды, аргументы, опции и прочие дополнения. Типичная операция, выполняемая в Docker, выглядит вот так:

|  |
| --- |
| docker container start |

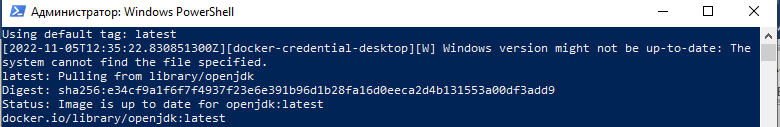
Например, если мы напишем следующую команду в Power Shell, она загрузит образ Java.

|  |
| --- |
| docker run openjdk |

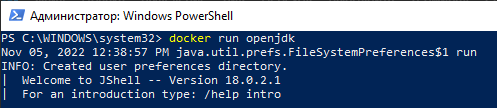


Как правило, образы создаются на основе базового — из Docker Hub, где есть множество уже готовых образов и которые ты можешь использовать, а не тратить время на создание собственного. Для загрузки образа используется команда docker pull.

|  |
| --- |
| docker pull openjdk |

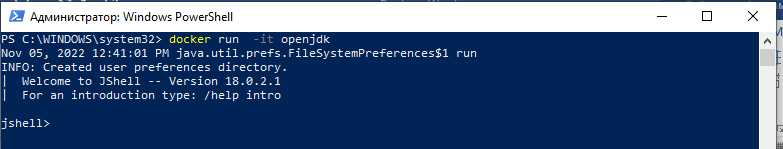


Запускает контейнер на основе указанного образа. Пример команды docker run openjdk В данном случае будет запущен контейнер из образа openjdk.



Для запуска образца в интерактивном формате используется следующая команда:

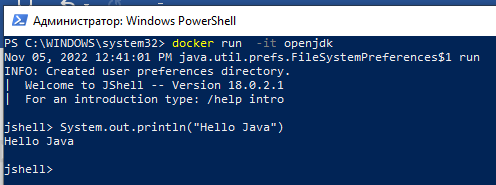
|  |
| --- |
| docker run -it openjdk |



В этом случае вы можете писать Java-команды в jsheell.

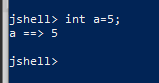
1. Hello Java

Чтобы отобразить простое сообщение «Hello Java», напишите команду печати без создания класса и нажмите Enter.

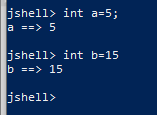


1. Переменные

Мы можем объявлять переменные и использовать их в любом месте сеанса Jshell. Создадим целочисленную переменную.



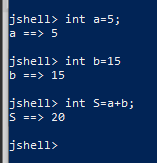
Точка с запятой (;) необязательна, мы можем оставить ее, и она отлично работает. Видите, переменная **b** создается без использования точки с запятой.



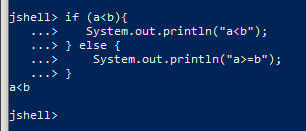
1. Выражения

Мы можем протестировать любое допустимое выражение Java, чтобы получить мгновенный результат.

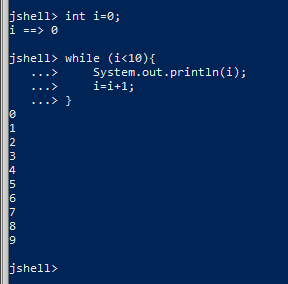
**Сложение двух целых чисел**



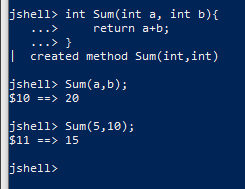
1. Пример оператора If-Else



1. Пример цикла while



1. Пример функция



**Преимущества JShell**

Jshell сократил все усилия, необходимые для запуска программы Java и тестирования бизнес-логики.

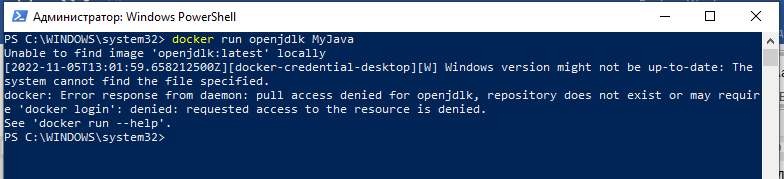
* Если мы не используем Jshell, создание Java-программы включает следующие шаги.
* Откройте редактор и напишите программу
* Сохраните программу
* Скомпилируйте программу
* Изменить, если есть ошибка времени компиляции
* Запустить программу
* Изменить, если какая-либо ошибка времени выполнения
* Повторите процесс

Jshell не требует вышеуказанных шагов. Мы можем оценивать операторы, методы и классы, даже можем написать приветственную программу без создания класса.

Для выхода из интерактивного режима нажимаем ctrl+D

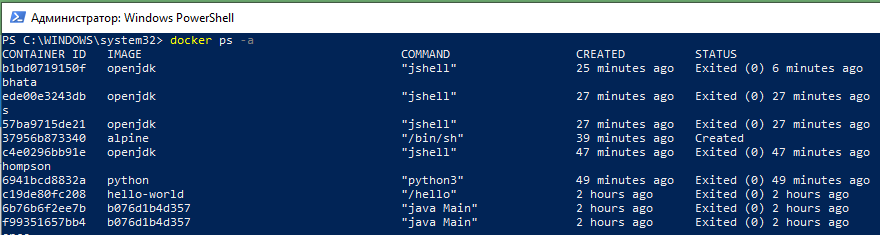
Мы можем дать псевдоним для нашего изображения.

|  |
| --- |
| docker run openjdlk MyJava |



**Остановить, запустить и назвать контейнеры**

Контейнеры Docker можно запускать, останавливать и перезапускать. Когда мы останавливаем контейнер, он не удаляется, но статус изменяется на остановлен, и процесс внутри контейнера останавливается. Когда мы запустили docker ps команду, вывод по умолчанию должен отображать только запущенные контейнеры. Если мы передаем –all или –a для краткости, мы увидим все контейнеры в нашей системе, независимо от того, запущены они или остановлены.



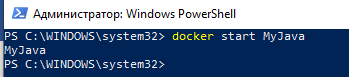
Команда docker start запускает контейнер, который существует, но находится в нерабочем состоянии.

Синтаксис следующий:

|  |
| --- |
| docker start [опции] название или ID контейнера [название или ID контейнера]. |

Все просто – вводите команду, а потом указываете название контейнера, чтобы его запустить:

docker start MyJava — пример старта через название MyJava



## ****Команды для управления контейнерами****

Контейнер — ключевая единица в Docker. Поэтому для начала разберемся, как управлять ей.

У описанных ниже команд есть общий синтаксис, который выглядит так: **docker container*название команды***. Названия, собственно, ниже:

* **create** — создает контейнер из выбранного образа.
* **start** — активирует уже существующий контейнер.
* **run** — создает новый контейнер и сразу включает его.
* **ls**— отображает все существующие контейнеры.
* **inspect**— подробнее рассказывает о выбранном контейнере.
* **logs** — выводит в консоль логи (то есть журнал событий).
* **stop**— пытается остановить выбранный контейнер, отправив ему сигнал **SIGTERM**, требующий завершить всю активность и сохранить пользовательские данные. Если ответ занимает слишком много времени, то следом посылает сигнал **SIGKILL**, чтобы «убить» процесс без сохранения данных.
* **kill**— выполняет ту же задачу, что и предыдущая команда, но пропускает шаг с отправкой **SIGTERM**. Сразу выключает контейнер, игнорируя сохранение пользовательских данных.
* **rm** — удаляет выбранный контейнер (он должен быть выключен, чтобы команда сработала).

**Далее рассмотрим каждую из команд подробнее.**

### **container create**

Из названия понятно, что речь идет о создании нового контейнера. Это основная функция Docker, с нее начинается его работа.

Синтаксис следующий:

|  |
| --- |
| **docker create [опции] *название образа* [дополнительные команды][аргументы]**. |

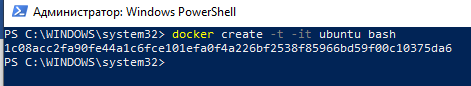
В квадратных скобках указываются опциональные компоненты команды. Использовать их необязательно, но можно оптимизировать создаваемый контейнер под свои задачи.

К примеру, добавить ссылку на другой контейнер с помощью тега **–link**. Или подключить **Standard Input** с помощью команды в духе:

|  |
| --- |
| docker create -a STDIN название образа |

Либо создать контейнер из образа операционной системы Ubuntu, добавив подключение к терминалу и запуск bash при каждом старте Docker:

|  |
| --- |
| docker create -t -it ubuntu bash |



Только при наличии терминала можно управлять контейнером. Поэтому его нужно обязательно запрашивать при каждому запуске.

### **container start**

В отличие от предыдущей команды, эта запускает контейнер, который существует, но находится в нерабочем состоянии.

Синтаксис следующий:

**docker start [опции] *название или ID контейнера* [название или ID контейнера]**.

Все просто – вводите команду, а потом указываете название контейнера, чтобы его запустить:

docker start 49920 — пример старта через ID контейнера 49920

docker start testing\_docker — пример старта через название testing\_docker

Какой из вариантов использовать, решает разработчик, ориентируясь на собственные предпочтения. Технических отличий нет.

### **container run**

Комбинирует две вышеописанные команды. Сначала создает новый контейнер, а потом тут же его включает.

Синтаксис следующий:

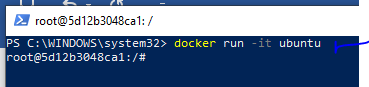
**docker run [опции] название образа [команды][аргументы]**.

Практика та же, что мы видели в синтаксисе команды **docker create**. Разница в одном слове и доступных опциях.

Чтобы создать контейнер из образа Ubuntu и тут же взять контроль над ним через командную строку, нужно сделать запрос терминала через опции команды.

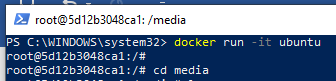
К примеру,

docker run -it Ubuntu

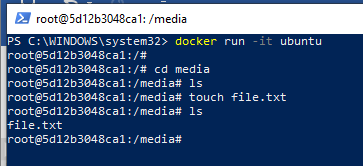


Используется для подключения к контейнеру с возможностью передавать ему команды после создания.

Здесь вы можете выполнить необходимые действия с ОС Ubuntu. Например переход на media



**Создание файла file.txt (touch file.txt) и отображение списка файлов (папок) - ls**



### **stop**

Выполняет противоположную функцию. То есть не запускает контейнер, а, наоборот, останавливает его работу.

Синтаксис следующий:

**docker stop [опции] название или ID контейнера [название или ID контейнера].**

Например, чтобы выключить контейнер с именем Timeweb, надо ввести в терминал:

docker stop timeweb

Аналогично с ID. Просто имя надо заменить на уникальный код, привязанный к контейнеру. Получится что-то в духе:

docker stop 14490

### **restart**

Последняя команда, связанная с рабочим состоянием. Она выполняет перезапуск выбранного контейнера.

Синтаксис следующий:

**docker restart [опции] название или ID контейнера [название или ID контейнера]**.

Вводите команду, а потом указываете название контейнера, чтобы его перезапустить:

docker restart 49920 — пример перезапуска через ID контейнера 49920

docker restart testing\_timeweb — пример перезапуска через название testing\_timeweb

### **container ls (или docker ps)**

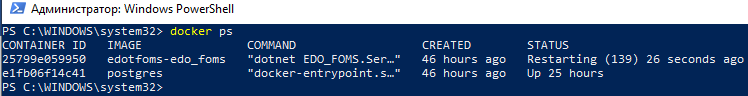
Обе команды выполняют одну и ту же задачу. Просто одна из них использовалась раньше, а другая появилась позднее.

**docker ps** отображает в терминале все запущенные контейнеры. А при добавлении опции **-a** в список попадают все контейнеры, созданные в системе.

В вывод команды попадают следующие параметры:

* **ID** — то есть кодовое значение конкретного контейнера.
* **IMAGE**— образ, используемый контейнером.
* **COMMAND** — список команд, которые должны выполняться сразу после запуска.
* **CREATED** — статус и время, когда контейнер бы создан (допустим, «35 минут назад»).
* **STATUS** — текущее состояние – включен ли и время работы с последнего запуска. Тут может отображаться код выхода и время, когда он произошел.
* **PORTS** — порты, используемые контейнером из списка.
* **NAMES** — более понятные в сравнении с ID имена, которые можно точно так же использовать для исполнения команд или поиска.

При желании можно использовать опцию **volume**, чтобы в терминале отображались только тома, являющиеся предпочтительным механизмом хранения данных в Docker.



### **logs**

Одна из команд, позволяющая подробнее ознакомиться с тем, как работает Docker. Точнее одна из его запущенных единиц. Показывает, как функционирует контейнер и что в текущий момент с ним происходит. Выводит некий журнал данных.

Если указать ключ **–follow** и название контейнера, то в терминал в реальном времени будут выводиться логи Docker. Так можно беспрерывно наблюдать за его работой и возможным появлением ошибок.

### **inspect (для контейнера)**

Отображает подробную информацию из Docker после обращения к конкретному контейнеру.

Синтаксис следующий: **docker inspect [опции] *название или ID контейнера* [название или ID контейнера]**.

По умолчанию отображает данные в формате JSON. Можно использовать дополнительную опцию **–format** для смены формата данных.

docker inspect –format=‘{{range .NetworkSettings.Networks}}{{.MacAddress}}{{end}}’ $INSTANCE\_ID — пример вывода информации о MAC-адресе

docker inspect –format=‘{{range $p, $conf := .NetworkSettings.Ports}} {{$p}} -> {{(index $conf 0).HostPort}} {{end}}’ $INSTANCE\_ID — пример вывода информации о портах

### **rm**

Команда удаления одного или нескольких контейнеров Docker, найденных на диске вашего хостинга.

Синтаксис следующий: **docker rm [опции] название или ID контейнера [название или ID контейнера]**.

Чтобы удалить сразу несколько контейнеров с разными названиями и ID, нужно ввести:

docker rm 45fs testing\_timeweb

Перед тем как удалить контейнер, нужно его выключить. Для этого надо воспользоваться командой:

docker stop

## ****Команды Docker для управления образами****

Еще одна значимая единица в Docker — образы. Управление ими во многом похоже на управление контейнерами, но есть ряд отличий, которые важно учитывать. Причем как в командах, так и в опциях.

У описанных ниже команд есть общий синтаксис, который выглядит так:**docker image *название команды***. Названия, собственно, ниже:

* **build** — собирает образ с нуля.
* **push**— отправляет образ в реестр.
* **pull** — загружает готовый образ с необходимыми для работы параметрами.
* **ls** — показывает все существующие образы.
* **history** — показывает каждый слой образа в ретроспективе, отображая ряд полезных сведений.
* **inspect** — рассказывает все, что известно об образе, включая данные, касающиеся отдельных слоев.
* **rm** — удаляет образ Docker из системы.
* **images** — списком показывает все образы Docker, найденные на диске.

Рассмотрим каждую из них немного подробнее.

### **build**

С помощью этой команды можно собрать образ, используя заранее подобранные параметры. Собираются одни из данных, хранящихся в файлах Dockerfile. Они находятся по определенным URL или заранее загружены в файловую систему.

Синтаксис следующий: **docker image build [опции] название и адрес файла Dockerfile, из которого будет собран образ.**

Например, если вы хотите собрать образ с названием Timeweb и тегом host в директории moy\_repositoriy, то надо ввести в терминал:

docker image build -t moy\_repositoriy/timeweb:host

Образами можно делиться через сервис Docker Hub. Тогда ими смогут пользоваться другие пользователи Docker, указав при сборке соотвествующий адрес. Для входа в Hub используется команда **docker login** (еще упомянем ее ниже).

### **push**

После сборки образа на диске хостинга его можно разместить в Docker Hub с помощью отдельной команды.

Синтаксис следующий: **docker image push [опции] название и адрес, где будет размещен образ**

К примеру, чтобы отправить в хаб образ Timeweb, который мы создали ранее, нужно ввести в терминал:

docker image push moy\_repositoriy/timeweb:host

Он будет размещен по адресу, указанному в команде **push**. После этого его смогут вытащить другие пользователи.

### **pull**

В Docker Hub находится большой список уже готовых образов, которые были созданы вами или другими пользователями. Чтобы не собирать новые из Dockerfile, можно скачать уже имеющиеся и работать с ними. Чтобы их вытащить, нужна команда **pull**.

Синтаксис следующий: **docker image pull [опции] название и адрес, где будет размещен образ.**

К примеру, чтобы вытащить из хаба образ Timeweb, который мы создали ранее, нужно ввести в терминал:

docker image pull moy\_repositoriy/timeweb:host

Аналогичным образом его смогут вытаскивать другие пользователи, обращаясь к адресу и названию образа, которые вы указывали.

### **ls**

**docker image ls** отвечает за отображение в терминале дополнительной информации об образах. Она показывает их список (как и в случае с контейнерами) и информацию о размере каждого.

Синтаксис следующий:**docker image pull [опции][адрес репозитория:тег]**.

В вывод команды можно добавить другие данные с помощью отдельных ключей:

**-a** — увеличивает список отображаемых образов за счет отображения даже промежуточных. **–digests** — добавляет в вывод команды еще и дайджесты. **-f** — фильтрует выдачу по заранее определенным условиям. **–format** — меняет формат отображения данных на основе шаблона Go. **–no-trunc**— отключает усечение информации в терминале. **-q**— показывает ID вместо названий образов.

### **history**

Показывает своего рода историю образа. Речь идет о «слоях», то есть промежуточных вариациях образа, входящих в его состав. Команда **history** показывает их названия, размер и дату создания.

Синтаксис следующий: **docker history [опции] *название образа.***

Например, чтобы отобразить слои образа Timeweb, введем в терминал:

docker image history timeweb

### **inspect (для образа)**

Отображает подробную информацию из Docker после обращения к конкретному образу.

Синтаксис следующий: **docker inspect [опции] *название или ID контейнера*[название контейнера или образа]**.

По умолчанию отображает данные в формате JSON. Можно использовать дополнительные опции **–format**для смены формата данных. Опция **-s** показывает размер инспектируемых элементов.

docker inspect –format=‘{{.Config.Image}}’ $INSTANCE\_ID — пример вывода информации о названии образа

docker inspect –format=‘{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}’ $INSTANCE\_ID — пример вывода информации об IP-адресе

### **rm**

Команда удаления одного или нескольких образов Docker, найденных на диске вашего хостинга.

Синтаксис следующий: **docker rm [опции] *название и путь до образа***.

Чтобы удалить образ Timeweb из системы, нужно ввести:

docker rm moy\_repositoriy/testing\_timeweb:host

Эта команда позволяет выбирать только образы, которые хранятся локально. Удаленные элементы из репозитория удалить не получится.

Чтобы удалить все локальные образы, обнаруженные в системе, введем в терминал:

docker image rm $(docker images -a -q)

### **images**

Команда показывает все образы Docker, которые расположены на жестком диске вашего сервера (или локально).

Вывод команды **docker images** выглядит так:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **REPOSITORY** | **TAG** | **IMAGE ID** | **CREATED** | **SIZE** |
| fedora | new | 990fqD00124 | 12 hours ago | 209MB |
| httpd | new | 24eaet001s0s | 24 hours ago | 152MB |
| ubuntu | recent | 30e102f4fas2 | 1 week ago | 179MB |
| fedora | recent | 92s0qF0D104 | 2 weeks ago | 133MB |
| ubuntu | new | 1d42e9ss40ft | 24 hours ago | 161MB |
| fedora | recent | eSS208s6das2 | 1 week ago | 190MB |
| mysql | tag | pq544s09efg0 | 38 hours ago | 452MB |

Здесь отображаются следующие данные, касающиеся образа:

**REPOSITORY** — это репозитория, в котором хранится образ. **TAG**— привязанный к нему тег (прописывается в конце команды после двоеточия. Например, **docker image build moy\_repositoriy/testing\_timeweb:host**. **IMAGE ID** — код образа, используемый вместо имени. **CREATED**— срок жизни. Показывает, как давно был создан образ. Неделю назад, 12 часов назад и т.п. **SIZE** — пространство, которое образ занимает на диске.

## ****Прочие команды****

Это общие операции, не привязанные к работе с конкретными сущностями типа образов и контейнеров. Но они тоже пригодятся при использовании Docker.

* **docker version** — показывает техническую информацию о самом Docker. Как о клиенте, так и о сервере.
* **docker login**— авторизует пользователя в реестре Docker.
* **docker system prune**— выполняет некую чистку, удалив из системы контейнеры, которые уже не используются, ненужные сети и образы без имен и меток.

### **version**

К этой команде можно обратиться за подробной информацией о самом Docker. Она отобразит версию клиента и сервера Docker.

### **login**

Об этой команде уже говорили. Она позволяет авторизоваться в реестре образов Docker Hub. Загружать туда образы и выгружать уже готовые к себе.

### **system prune**

По умолчанию эта команда удаляет только данные без тегов, но можно настроить ее так, чтобы из системы были удалены все неиспользуемые элементы.

Допустим, чтобы стереть из Docker все образы и сверху зацепить еще тома, то надо ввести в терминал:

docker system prune –all –volumes

На этом все. Это даже больше того минимума, который нужен для работы с Docker на начальных этапах. Этих команд хватит, чтобы собрать образ, запустить контейнер и перейти к разработке. А уже по ходу дела вы сможете обращаться в документацию и обогащать свои знания.

Dockerfile и свои образы

Выполнять команды в терминале не столь удобно, особенно когда речь идет про подключение образа с большим количеством настроек. В ходе урока мы с вами научимся работать с файлом Dockerfile. Он позволяет описать характеристики подключаемого образа.

Терминал удобен для выполнения небольших команд. Когда кода становиться слишком много, то всегда проще поместить его в некий файл, а позже выполнить. В Docker вы можете помещать описание образов в отдельный файл, что называется Dockerfile.

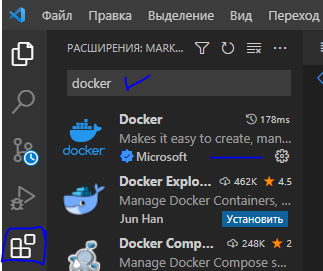
**Основные команды Dockerfile**

Dockerfile состоит из нескольких основных команд:

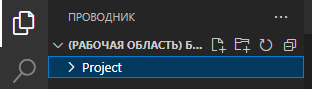
* FROM – указывает какой образ вы будете стягивать (скачивать). Тут же можно указать версию образа
* WORKDIR – указывает рабочую папку
* COPY – указывает какие файлы из вашего проекта будут скопированы на хост машину. Другими словами, указывает какие файлы будут выполнены за счет возможностей образа
* EXPOSE – указывает порт для проекта. Будет работать в том случае, если в образе есть локальный сервер
* RUN – позволяет описать команду, что выполниться один раз при сборке всего контейнера
* CMD – описывает команду, что выполняется каждый раз при старте контейнера

Есть и другие команды, но представленные выше используются, практически, в каждом Dockerfile.

1. Откройте Visual Code и установите докер и удаленные контейнеры (Dev Containers).



1. Создайте папка с названием Project



1. Внутри папки Project создайте файл с именем Dockerfile перепишите следующие код.

|  |
| --- |
| FROM openjdk:11  COPY . /java  WORKDIR /java  RUN javac Main.java  CMD ["java", "Main"]  EXPOSE 8001 |

1. Внутри папки Project создайте файл с именем Main.java и перепишите следующие код.

|  |
| --- |
| class Main {      public static void main(String[] args) {          System.out.println("Hello World!");      }  } |

**Построение файла**

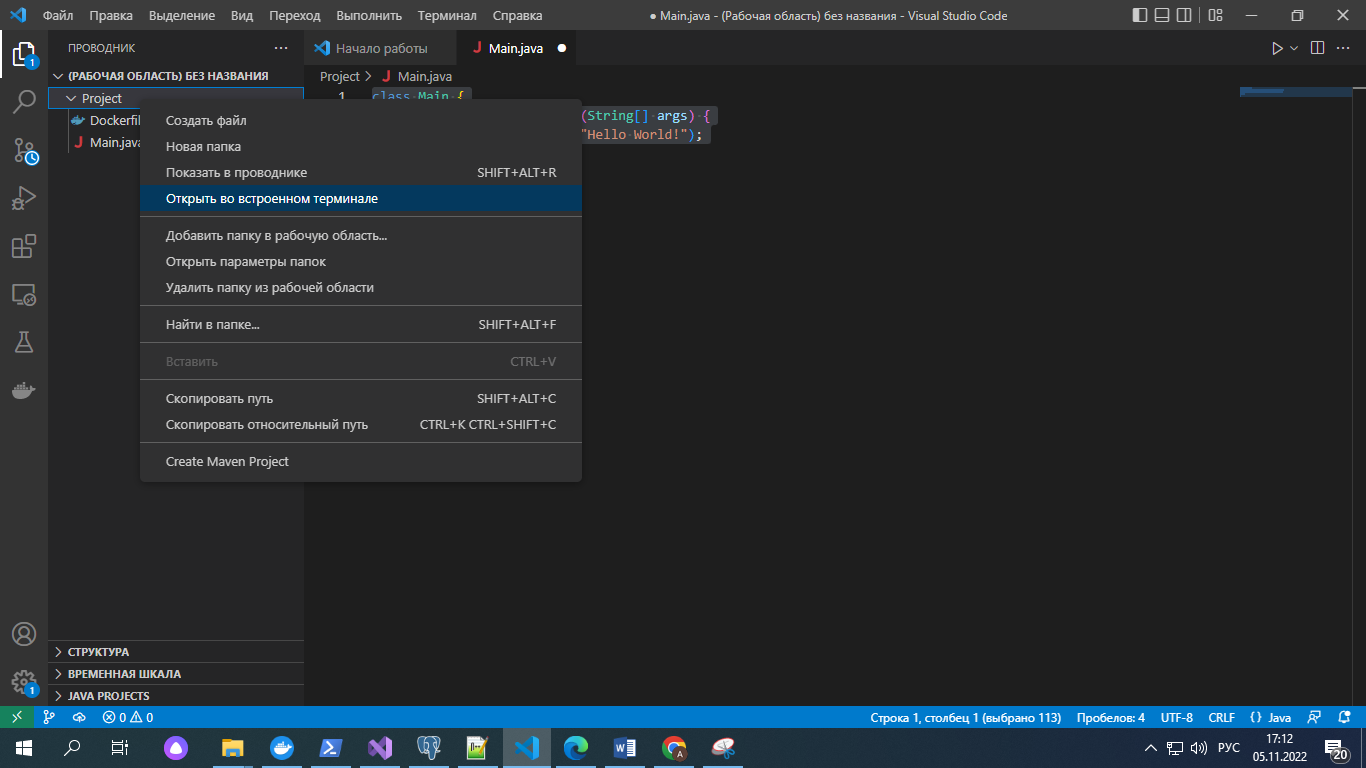
Для построения файла вам нужно прописать команду в терминале:

docker build .

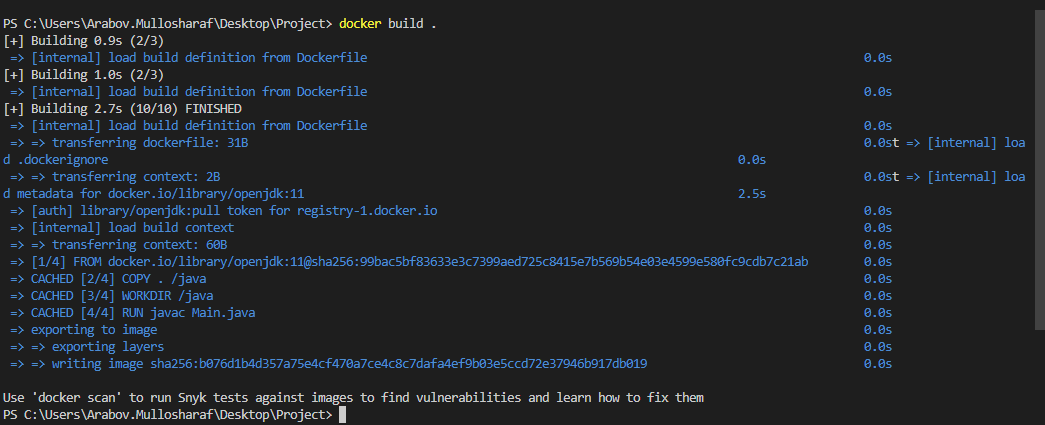
Здесь же вместо точки должен идти полный путь к файлу Dockerfile. Для запуска проекта используйте команду run и ID вашего созданного контейнера:

docker run ID

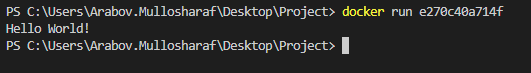
1. Щелкните правой кнопкой мыши на папке Project и выберите команду "Открыть во встроенном терминале".



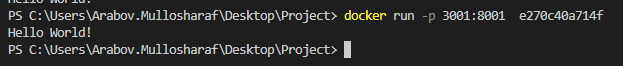
1. В открывшемся терминале пишем команду docker build . и нажимаем Enter.



1. 7) С помощью команды docker images найдите только что созданный образ и скопируйте его ID. Обычно он имеет вид <noname>
2. Запускаем проекта



1. Запуск с помощью порта



**Другие команды**

#1. docker ps — смотрим список запущенных контейнеров

Наверное, следует начать с команды docker ps, выводящей список запущенных контейнеров. Ей можно передать несколько параметров, вот самые полезные из них:

-q — «тихий» режим, в котором команда выводит только id контейнеров (полезно, когда вам нужно знать только id или же при использовании этой команды в сценариях).

-a — показывает все контейнеры, а не только запущенные.

#2. docker pull — загрузка образа

Как правило, образы создаются на основе базового — из Docker Hub, где есть множество уже готовых образов и которые ты можешь использовать, а не тратить время на создание собственного. Для загрузки образа используется команда docker pull.

#3. docker build — собирает образ

Данная команда собирает образ Docker из файла докера (dockerfile) и контекста сборки. Контекст сборки — это набор файлов, расположенных по определенному пути. Для задания имени образа используйте параметр -t, например, «docker build -t my.». Собирает образ из текущего каталога (».«) — последний параметр это имя каталога, в нашем случае точка указывает, что каталог — текущий.

#4. docker logs — смотрим логи

Позволяет просмотреть логи указанного контейнера. Можно использовать флаг -follow, чтобы следить за логами работающего контейнера, например, docker logs -follow my.

#5. docker run — запускаем контейнер

Запускает контейнер на основе указанного образа. Пример команды docker run my -it bash В данном случае будет запущен контейнер из образа my, а после в нем будет запущен bash.

#6. docker stop — останавливает контейнер

Используется для «мягкой» остановки контейнера. Пример: docker stop my\_cont. Можно остановить не конкретный контейнер, а все запущенные — docker stop $(docker ps -a -q).

#7. docker kill — «убивает» контейнер

Не пытается аккуратно завершить процесс, подобна системной команде kill. Как и в предыдущем случае, можно «убить» все контейнеры: docker kill $(ps -a -q).

#8. docker rm — удаляет контейнер

Для удаления контейнера используется команда docker rm, например, docker rm my\_cont.

#9. docker rmi — удаляет образ

Команда docker rmi (i от image) удаляет образ, например, docker rmi my.

#10. docker volume ls — список томов

Данная команда показывает список томов, которые являются основным механизмом для хранения данных, генерируемых контейнерами Docker.

Docker compose

Ранее мы все время устанавливали лишь по одному образу в проект. За счет Docker Compose вы можете устанавливать множество образов, указывать связь между ними и добавлять к ним характеристики. В ходе урока мы научимся работать с Докер Compose.

Что такое Docker Compose?

Docker Compose представляет из себя отдельный файл, в котором вы можете описать все характеристики относительно образов. Вы можете указать какие образы будут подключены, какие у них будут характеристики, возможности и так далее.

Огромным преимуществом Docker Compose является возможность подключения сразу многих образов. Так, вы можете подключить языки Джава, Python, C++, PHP и к ним дополнительно настроить сервера, среды разработки и все это в одном файле.

При построении файла Compose вы указываете сервисы (выполняют роль образов) и указываете их характеристики.

Сборка проекта

После описания проекта вам необходимо его собрать. Для этого используется команда docker-compose build. Сама команда  еще ничего не запускает, но зато собирает все сервисы для дальнейшего их запуска.

Для старта проекта прописывается другая команда: docker-compose up. Она служит для «поднятия» всех сервисов. Другими словами, для их запуска.

Противоположная команда docker-compose down наоборот «опускает» все сервисы, останавливая ваш проект.

Практика

1. В Visual Code создайте папку с именем php, внутри нее создайте файл с именем index.php и напишите в нем следующий код:

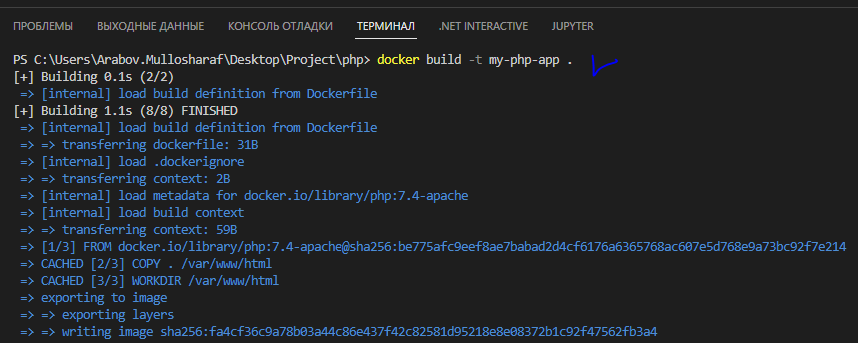
|  |
| --- |
| <?php  echo 'Hello World!'  ?> |

1. В парке php создайте файл Dockerfile со следующее содержание

|  |
| --- |
| FROM php:7.4-apache  COPY . /var/www/html  WORKDIR /var/www/html  CMD [ "php", "./index.php" ]  EXPOSE 80 |

1. В терминале напишите следующую команду и нажмите Enter

docker build -t my-php-app .



1. Запускаем проекта: docker run -p 8001:80 -t my-php-app



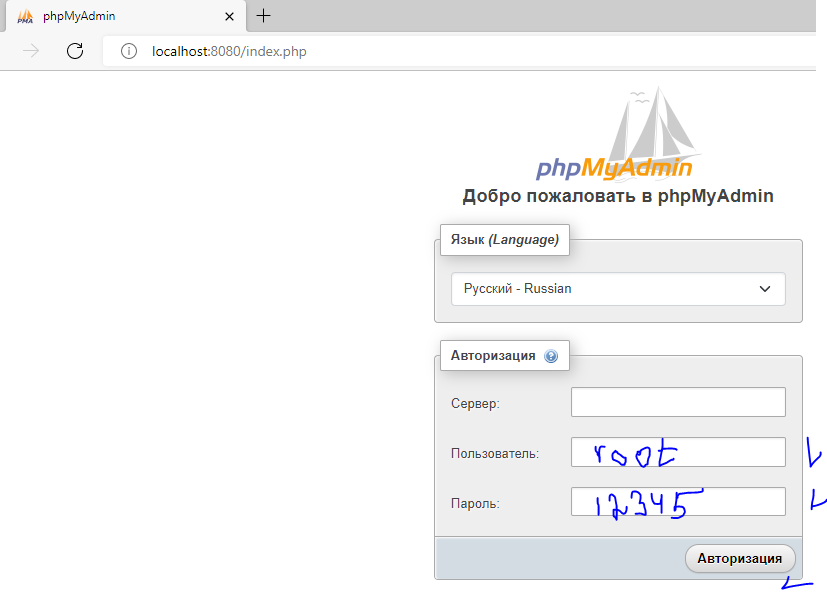
1. Создайте файл с именем docker-compose.yml в папке php
2. Заходим на https://hub.docker.com/ и набираем в поиске phpmyadmin и копируем код из секции " Usage with docker-compose and arbitrary server" в файл docker-compose.yml.

|  |
| --- |
| version: '3.1'  services:    db:      image: mariadb:10.6      restart: always      environment:        MYSQL\_ROOT\_PASSWORD: notSecureChangeMe    phpmyadmin:      image: phpmyadmin      restart: always      ports:        - 8080:80      environment:        - PMA\_ARBITRARY=1 |

1. В разделе db добавим следующего кода:

|  |
| --- |
| MYSQL\_ROOT\_PASSWORD: 12345 |

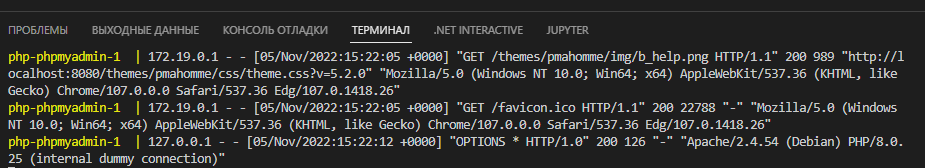
1. docker-compose build
2. docker-compose up
3. Теперь вы можете войти в phpmyadmin, набрав http://localhost:8080/index.php в браузере.



1. Теперь мы используем mySQl вместо mariadb:10.6. Для этого введите следующие команды в файл docker-compose.yml.

|  |
| --- |
| version: '3.1'  services:    db:      image: mysql      restart: always      command: --default-authentication-plugin=mysql\_native\_password      environment:        MYSQL\_ROOT\_PASSWORD: 12345    adminer:      image: adminer      restart: always      ports:        - 8080:8080    phpmyadmin:      image: phpmyadmin      restart: always      ports:        - 8080:80      environment:        - PMA\_ARBITRARY=1 |

1. Нажмите ctrl+C, чтобы выйти из режима php в терминале.



1. docker-compose build
2. docker-compose up
3. Для запуска php в разделе services файла docker-compose.yml перепишем следующие команды

|  |
| --- |
| services:    php:       build: ./php       ports:         - 8081:80 |

Т.е.

|  |
| --- |
| version: '3.1'  services:    php:       build: ./php       ports:         - 8081:80    db:      image: mysql      restart: always      command: --default-authentication-plugin=mysql\_native\_password      environment:        MYSQL\_ROOT\_PASSWORD: 12345    adminer:      image: adminer      restart: always      ports:        - 8080:8080    phpmyadmin:      image: phpmyadmin      restart: always      ports:        - 8080:80      environment:        - PMA\_ARBITRARY=1 |

1. docker-compose build
2. docker-compose up
3. Переходим по адресу <http://localhost:8081/> и получим Hello World в браузере.