DOCKER GUIDE FOR EVERYONE

By: Pavan Belagatti

TABLE OF CONTENTS

INTRODUCTION

Containers have become mainstream in the software development world. Today, every company uses containers to fasten their development life cycle and easy collaboration between the teams, fostering the DevOps principles within the organization.

Consider the following scenario:
A developer completes a feature or bug fix for an application and then delivers a build for testing. But, the development and testing systems are different; this results in the application behaving in unexpected ways. Such a situation creates havoc between teams and degrades the working atmosphere, affecting the productivity and the morale of development and operations teams.

Docker has made working with containers very easy and helps developers and operations work as one team. Docker helped to bridge the gap between Dev and Ops that has existed for many years.

EVOLUTION OF DOCKER

Long ago, before the introduction of Docker and containers, big firms would go and buy many servers to make sure their services and business would keep running, uninterrupted. This process usually meant that firms bought more servers than needed, which was extremely expensive. But they needed to do this because, as more and more users hit their servers, they wanted to make sure they could scale well without any downtime or outage.

VMware and IBM entered the scene (there is still a debate on who introduced it first) and introduced Virtualization that allowed for running multiple operating systems on the same host. This was a game-changer, but still came with the high price tag of multiple kernels and OSs. So fast forward to modern-day containerization, the company 'Docker' solves a lot of problems.

WHAT IS DOCKER

Docker is a boon to the software industry. It is a platform that helps DevOps teams worldwide to automate the deployment of applications. Docker does it by simplifying the process of building, running, managing, and also distributing applications by virtualizing the operating system of the computer on which it is installed and running. The aim is to make applications work efficiently in different system environments.

WHY DOCKER

Docker changed the way applications are built and shipped. It has completely revolutionized the containerization world. With Docker, deploying your software becomes a lot easier; you don't have to think about missing a system configuration, underlying infrastructure, or a prerequisite. Docker enables us to create, deploy, and manage lightweight, stand-alone packages that contain everything that is needed to run an application. To be specific, it contains all of the required code, libraries, runtime, system settings, and dependencies. These packages are called containers. Each container can be deployed with its own portion of the host's CPU, network resources, memory, and everything. Containers can share the host's kernel and operating system.

Imagine you've thousands of test cases to run connected to a database, and they all go through sequentially. How much time do you think that will take? Using Docker, these test cases can run much more quickly. A containerization approach can allow the tests to run in parallel on the same host at the same time.

BENEFITS OF USING DOCKER

- Enables consistent environment
- Easy to use and maintain
- Efficient use of the system resources
- Increase in the rate of software delivery
- Increases operational efficiency
- Increases developer productivity

Learn more in my original article: '[Getting Started with Docker: Facts You Should Know]'.

DOCKER ARCHITECTURE

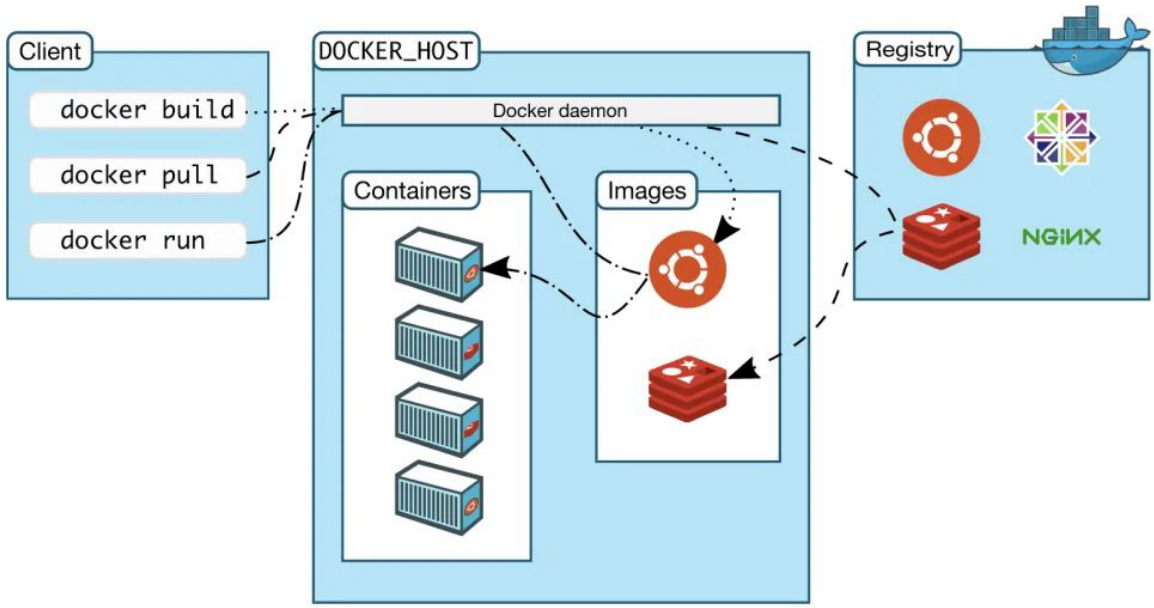


Image Source Credits: ***Docker***

Docker works on a client-server architecture. There is something called Docker client that talks to the Docker daemon, which does the heavy lifting of building, running, and distributing Docker containers. Both the Docker client and Docker daemon *can* run on the same system, or the other way is, you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, through UNIX sockets or network interface.

MAIN COMPONENTS OF THE DOCKER SYSTEM

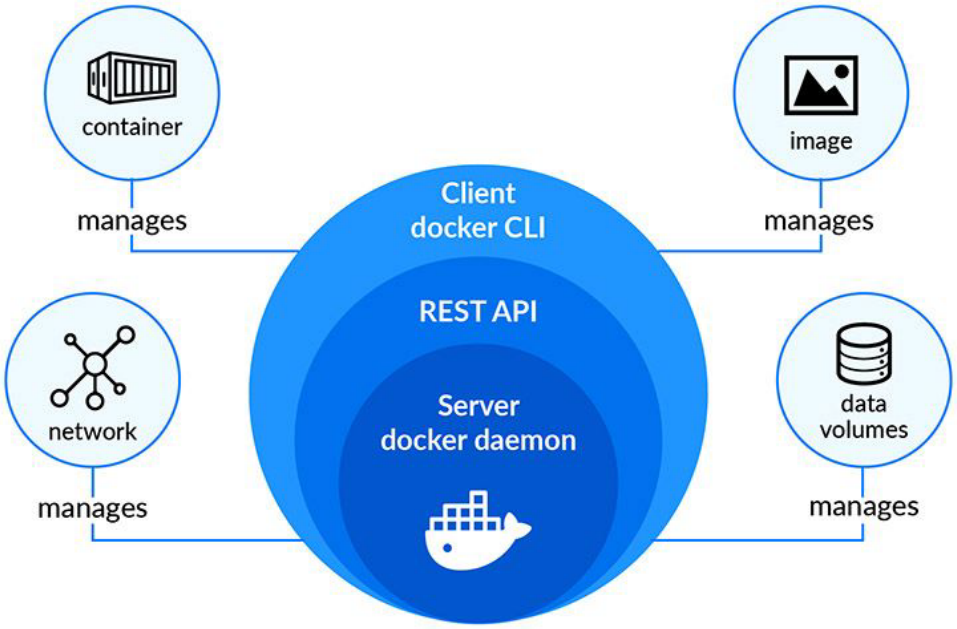


*Image source: **ThinkPalm***

- ***Docker Image***
  Docker Images are made up of multiple layers of read-only filesystems. These filesystems are described and created using a Dockerfile, which is just a text file with a set of pre-written commands.
  For every line of text written or instruction given in a Dockerfile, a layer is created and is placed on top of another layer forming a docker image, which will ultimately be used to create a docker container.
  Docker has created a public registry, hub.docker.com, where people can store their docker images. Docker images can also be stored in a local, private registry.

- ***Docker Container***
  A container is an isolated application, built from one or more images, and acts as an entire packaged system which includes all the libraries and dependencies required for an application to run. Docker containers come without an OS. They use the Host OS for functionality, hence it is a more portable, efficient and lightweight system that comes with a guarantee that the software will run in any environment.

- ***Docker Engine***
  It is the nucleus of the Docker system, an application that is installed on the host machine that follows client-server architecture.

There are 3 main components in the Docker Engine:
Server is the docker daemon named dockerd. Creates and manages docker images, containers, networks, etc.
Rest API instructs docker daemon what to do.
Command Line Interface (CLI) is the client used to enter docker commands.

- ***Docker client***
  Docker client is the key component in the Docker system which is used by users to interact with Docker via a command-line interface (CLI). When we run the docker commands, the client sends these commands to the daemon 'dockerd', to build, run, and stop the application.

- ***Docker Registry***
  This is the place where Docker images are stored. Docker Hub and Docker Cloud are public registries that can be accessed by everyone and anyone, whereas, anther option is having your own private registry. Docker by default is configured to look for images on Docker Hub. You can also have an Artifactory Docker Registry for more security and optimization of your builds.

- ***Image:*** Image is basically an executable package that has everything that is needed for running applications, which includes a configuration file, environment variables, runtime constraints, and libraries.
- ***Dockerfile:*** This contains all the instructions for building the Docker image. It is basically a simple text file with instructions to build an image. You can also refer to this as the automation of Docker image creation.
- ***Build:*** Creating an image snapshot from the Dockerfile.
- ***Tag:*** Version of an image. Every image will have a tag name. If a tag is not explicitly set when using the 'docker image tag' command, the tag of 'latest' will be applied.
- ***Container:*** A lightweight software package/unit created from a specific image version.
- ***DockerHub:*** Image repository where we can find different types of images.
- ***Docker Daemon:*** Docker daemon runs on the host system. Users cannot communicate directly with Docker daemon but only through Docker clients.
- ***Docker Engine:*** The system that allows you to create and run Docker containers.
- ***Docker Client***: It is the chief user interface for Docker in the Docker binary format. Docker daemon will receive the docker commands from users and authenticates the communication with Docker daemon.
- ***Docker registry***: Docker registry is a solution that stores your Docker images. This service is responsible for hosting and distributing images. The default registry is the Docker Hub.

Docker, as a tool, fits perfectly well in the DevOps ecosystem. It is built for the modern software firms that are keeping pace with the rapid changes in technology. You cannot ignore Docker in your DevOps toolchain; it has become a de facto tool and almost irreplaceable.
The things that make Docker so good for DevOps enablement are its use cases and advantages that it brings to the software development process by containerizing the applications that support the ease of development and fast release cycles.

Docker can solve most of the Dev and Ops problems, and the main one, 'It works on my machine,' enables teams to both collaborate effectively and work efficiently.
According to RightScale 2019 State of the Cloud Report, Docker is already winning the container game with an amazing YoY adoption growth.

With Docker, you can make immutable dev, staging, and production environments. You will have a high level of control over all changes because they are made using immutable Docker images and containers. You can always roll back to the previous version at any given moment if you want to.

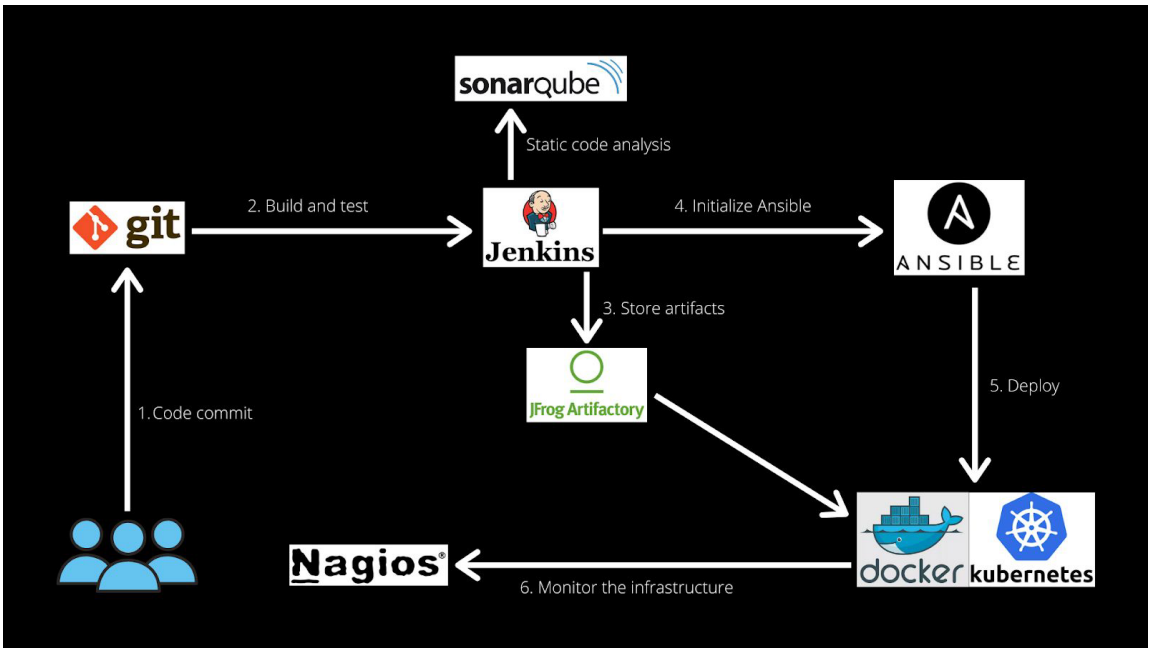
Development, staging, and production environments become more alike. With Docker, it is more likely that if a feature works in the development environment, it will work in staging and production, too.
Datadog took a sampling of its customer base, representing more than 10,000 companies and 700 million containers, in its report on the survey, it is shown that, at the beginning of April 2018, 23.4 percent of Datadog customers had adopted Docker, up from 20.3 percent one year earlier. Since 2015, the share of customers running Docker has grown at a rate of about 3 to 5 points per year.

Docker with Kubernetes

Docker helps to “create” containers, and Kubernetes allows you to “manage” them at runtime. Use Docker for packaging and shipping the app. Employ Kubernetes to deploy and scale your app. Startups or small companies with fewer containers usually can manage them without having to use Kubernetes, but as the companies grow, their infrastructure needs will rise; hence, the number of containers will increase, which can be difficult to manage. This is where Kubernetes comes into play.

When used together, Docker and Kubernetes serve as digital transformation enablers and tools for modern cloud architecture. Using both has become a new norm in the industry for faster application deployments and releases. While building your stack, it is highly recommended to understand the high-level differences between Docker and Kubernetes.
Let containers help to unveil the mysteries of cloud computing regardless of the cloud journey you choose.

Let’s take a simple scenario of a CI/CD setup using Docker and Kubernetes:



1. The developers’ code is pushed into the Git repository.
2. The build and test happen with Maven in Jenkins.
3. Using Ansible as a deployment tool, we will write Ansible playbooks to deploy on AWS.
4. We will introduce JFrog Artifactory as the repository manager after the build process from Jenkins; the artifacts will be stored in Artifactory.

[Note: The build process itself can use Artifactory as a proxy and/or cache for any required third party dependencies.]

5. Ansible can communicate with Artifactory, take the artifacts and deploy them onto the Amazon EC2 instance.
6. SonarQube can help in reviewing the code by providing static code analysis.
7. We then introduce Docker as a containerization tool. Just like the way we did on Amazon EC2, we will deploy the app in Docker containers by creating a Dockerfile and Docker images.
8. Once this above setup is done, we will introduce Kubernetes to create a Kubernetes cluster, and by using Docker images, we will be able to deploy.
9. Finally, we will use Nagios to monitor the infrastructure.

Learn more in my original article about how Docker and Kubernetes work together.

Steps to install Docker

Below is the link for docker installation provided by Docker. There is clear documentation available for installing docker on your chosen platform.

https://docs.docker.com/engine/install/

Once docker is installed, check the version of docker

```
root@gcp-slave-instance:~# docker --version
Docker version 19.03.8, build afacb8b7f0
root@gcp-slave-instance:~#
```

Few basic Docker commands

- **docker ps**
  Gives you the list of active containers on your machine

```
root@gcp-slave-instance:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
39dee19e7b38        httpd               "httpd-foreground"  47 seconds ago      Up 45 seconds       80/tcp              serene_joliot
root@gcp-slave-instance:~#
```

In the output we can see it displays several details about the container

CONTAINER ID: Each and every container will be assigned a unique ID

IMAGE: Every Image has an attached tag

Note:

If the image comes from Docker Hub, it will show [REPOSITORY:TAG], but if it is tagged with 'latest', it will only show [REPOSITORY].

If it's an image that comes from some other registry other than Docker Hub, then it will display [REGISTRY/REPOSITORY:TAG].

Otherwise, if it's an image you've built and it hasn't been tagged at all yet, it will display as [IMAGE ID].

COMMAND: The command specified in the Dockerfile that will be executed by the container when it is launched.

CREATED: Shows the detail when it was created

STATUS: Shows how long the container has been running

PORTS: Exposed port(s), if any

NAMES: Random name that is assigned by docker for container created unless you specify a name when launching the container

- **docker ps -a**
  Gives you the full list of containers including the ones which are stopped or crashed

```
root@gcp-slave-instance:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
0cece606a4d9        mysql               "docker-entrypoint.s…"   2 minutes ago       Exited (1) 2 minutes ago                        hopeful_sanderson
39dee19e7b38        httpd               "httpd-foreground"       4 minutes ago       Exited (0) 36 seconds ago                       serene_joliot
root@gcp-slave-instance:~#
```

- **docker images**
  Gives you the list of images present in the system

```
root@gcp-slave-instance:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
mysql               latest              8e8c6f8dc9df        24 hours ago        546MB
httpd               latest              bdc169d27d36        2 days ago          166MB
root@gcp-slave-instance:~#
```

- **docker run ARGUMENT IMAGE-NAME**
  It will create a container using the image name

```
root@gcp-slave-instance:~# docker run -itd httpd
39dee19e7b3892b338408e96b4c6630d0b30b68392b0c85a8db0efec8641b92d
root@gcp-slave-instance:~#
```

Here arguments -itd (or -i, -t, and -d) means

i – Interactive

t – Connected to terminal

d – Detached Mode

We can run the container using whatever arguments are required for our purposes.

- **docker stop CONTAINER-ID/NAME**
  To stop the container

```
root@gcp-slave-instance:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
c242f3177c7a        httpd               "httpd-foreground"   4 minutes ago       Up 4 minutes        80/tcp              stoic_jones
root@gcp-slave-instance:~# docker stop httpd
Error response from daemon: No such container: httpd
root@gcp-slave-instance:~# docker stop c242f3177c7a
c242f3177c7a
root@gcp-slave-instance:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
root@gcp-slave-instance:~#
```

- **docker rm CONTAINER-ID/NAME**
  To remove a container

```
root@gcp-slave-instance:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
39dee19e7b38        httpd               "httpd-foreground"   17 minutes ago      Exited (0) 13 minutes ago                       serene_joliot
root@gcp-slave-instance:~# docker rm serene_joliot
serene_joliot
root@gcp-slave-instance:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
root@gcp-slave-instance:~#
```

- **docker rmi IMAGE-ID**
  To remove a docker image

```
root@gcp-slave-instance:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
httpd               latest              bdc169d27d36        2 days ago          166MB
root@gcp-slave-instance:~# docker rmi bdc169d27d36
Untagged: httpd:latest
Untagged: httpd@sha256:d5dc0d279039da76a8b490d89a5c96da83a33842493d4336b42ccdfbd36d7409
Deleted: sha256:bdc169d27d36e2438ec8452c7dd7a52a05561b5de7bef8391849b0513a6f774b
Deleted: sha256:6535aa332fb72ca508f550fef8ffb832d4c6bc72a48720b42659e10d47668181
Deleted: sha256:c7bce1fab718a11501a672c895a729b1fdf8099d00fe152bef8c2534ee455976
Deleted: sha256:75b6b2392924b062257ed97e5c2f3aa9f50a922b94c3f7c342d0aed2370e8bec
Deleted: sha256:267e2020b1bd0b182eb02d1a0f3e2f72efc542890ef6159ed9c3570322608de0
Deleted: sha256:b60e5c3bcef2f42ec42648b3acf7baf6de1fa780ca16d9180f3b4a3f266fe7bc
root@gcp-slave-instance:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
root@gcp-slave-instance:~#
```

- **docker exec -it container name /bin/bash**

  Get access to shell of container
  With this command we can run our required code within the container.

```
root@gcp-slave-instance:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
20abc7597c02        httpd               "httpd-foreground"   2 minutes ago       Up 2 minutes        80/tcp              adoring_proskuriakova
root@gcp-slave-instance:~# docker exec -it 20abc7597c02 /bin/bash
root@20abc7597c02:/usr/local/apache2#
```

Containers are dynamic in nature. They move a lot. Today a container might be on server A, tomorrow it may be on server B, so they will be shuffled and relocated as required. A Docker volume acts as a data warehouse, or data storage attached externally to container. Let's say for example, in the data directory (data) we have data1 and data2 as files that are part of the data associated for our application to work with

```
root@gcp-slave-instance:~# mkdir data
root@gcp-slave-instance:~# ls
data
```

but we do not want these files to actually be stored within the container; instead of that, we want the data directory to be mounted, so the actual read and write will happen within data1 and data2 and it will look like they are part of the container. Directories mounted like this are called Volumes.

```
root@gcp-slave-instance:~# docker run -itd --name httpd-container -v /root/data:/root/files httpd
79d4114c3523d36147337346d6943c5839d0531ec99299b382426cd9a17c5191
root@gcp-slave-instance:~# docker ps
CONTAINER ID        IMAGE               COMMAND              CREATED             STATUS              PORTS               NAMES
79d4114c3523        httpd               "httpd-foreground"   15 seconds ago      Up 13 seconds       80/tcp              httpd-container
```

```
root@867ebcb500f9:/# cd /root
root@867ebcb500f9:~# ls
files
root@867ebcb500f9:~# cd files
root@867ebcb500f9:~/files# ls
data1  data2
```

Now that we have a basic idea about creating, deleting and starting a container, further will see how to create your own image.

On our server machine we need to install Apache2 by running below mentioned commands:

sudo apt-get update

sudo apt-get install apache2

CREATING A DOCKERFILE

Create a directory called ‘sample-code’. Within that directory, create a Dockerfile and index.html file.

```
root@gcp-slave-instance:~# ls
data  sample-code
root@gcp-slave-instance:~# cd sample-code
root@gcp-slave-instance:~/sample-code# ls
1  Dockerfile  index.html
root@gcp-slave-instance:~/sample-code#
```

```
FROM httpd
ADD index.html /user/local/apache2/htdocs/index.html


~
```

Every Dockerfile starts with the FROM command which tells where the base image is coming from. Here,we are using httpd as our base image. Next,we want to add a file index.html, which will act as our source, and our destination will be /usr/local/apache2/htdocs/index.html. When we run this file, docker will create an image out of it. Once the image is created we can use this image to create a container.

docker build -t FIRST-IMAGE: this command will build an image where FIRST_IMAGE is the name of image

docker run -itd --name FIRST_CONTAINER -p 8090:80 FIRST_IMAGE: this command will build the container where FIRST_CONTAINER is name of container mapped to port 80

we can see the output : http://server_IP:port

Learn more about Docker practicalities in my original article ‘Getting started with Docker’.

DOCKER USE CASES

Let us see some successful Docker use cases.

Spotify

This digital music service company, with millions of users, runs a microservices architecture with as many as 300 servers for every engineer on staff. Spotify struggled to manage the deployment pipeline with so many microservices. It then turned to a highly sophisticated platform like Docker to help developers pass the same container through their CI/CD pipeline for easy delivery.
From build to test to production, they ensured that the container that passed the build and test process was the same in production.
After using Docker, the company guaranteed that all of their services remain up and running throughout, providing a great user experience for their customers. They even built their new platform called 'Helios' based on Docker containers to deploy their containers across their entire fleet and maintain their development ecosystem.

ADP

ADP is one of those companies that keep using Docker to manage their application infrastructure better. ADP is the largest global provider of cloud-based human resources services. ADP handles HR for more than 600,000 clients from payroll to benefits, which caused a challenge in terms of security and scalability. To solve the security issue, ADP uses Docker Datacenter. Docker Content Trust enables their IT ops team to sign images and ensure that only signed binaries run in production. They also perform automated container scanning. Using multiple Docker Trusted Registries enables them to build an advanced trust workflow for their application development process.

ING

As one of the top 10 financial global leaders, ING is widespread with thousands of developers working in coordination and collectively.
Ing's IT organization in the Netherlands alone has more than 1,800 people. It created unprecedented challenges of coordinating change across large groups of people, methodologies, processes, and technology, leading to low-quality software.
Then ING thought of Docker as a better tool to align its IT organization and mitigate the challenges faced. After using Docker, ING can move faster with their CD pipeline running in Docker containers. The increased level of automation helped them to release quality software quickly and with more efficiency.

PayPal

PayPal started using Docker to help them achieve high availability, performance, and new operational efficiencies. It also helped them with a 50% increase in the speed of their developers' build-test-deploy cycles when developing and testing locally. PayPal also increased application availability through Docker's dynamic placement capabilities and improved security by using Docker to automate and granularly control access to resources. Docker usage at PayPal empowered developers to innovate and try new tools and frameworks, thereby increasing the overall productivity and team efficiency.

Source credits:
https://dzone.com/articles/top-10-benefits-you-will-get-by-using-docker

https://hackernoon.com/what-is-docker-and-benefits-of-docker-re2d32jh

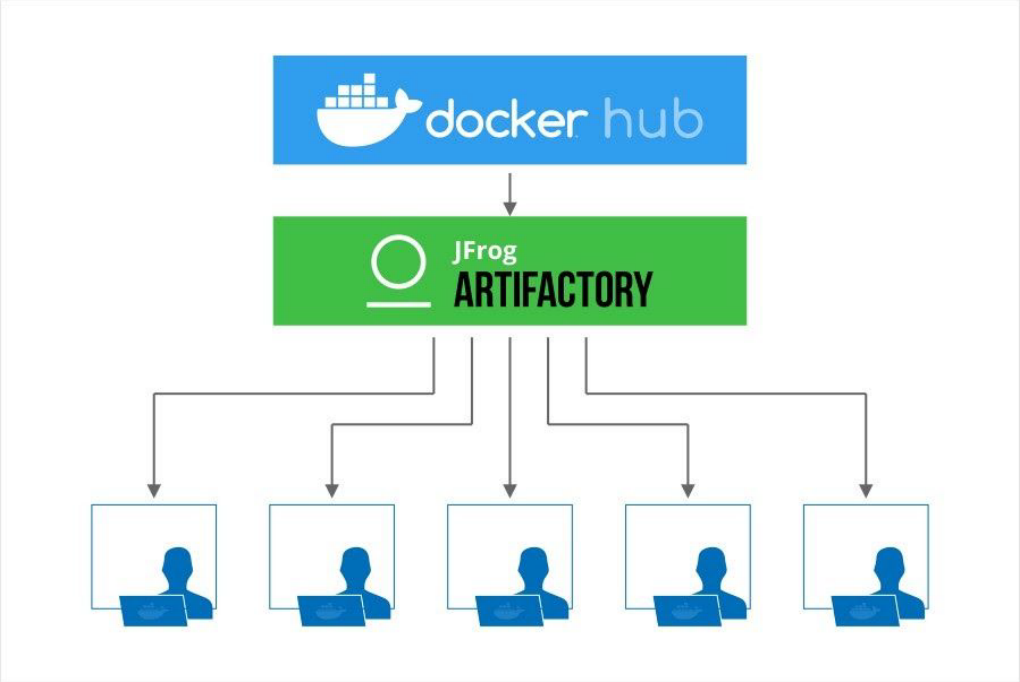
DOCKER BEST PRACTICES

Before approaching Docker, you must know some best practices to reap the benefits of this tool to the fullest extent. Listed here are some Docker best practices to keep in mind,

- Build images to do just one thing (Also, See Security Best Practices for Docker Images)
- Use tags to reference specific versions of your image
- Prefer minimalist base images
- Use multi-stage builds
- Don't use a root user, whenever possible
- Use official, purpose-built images
- Enable Docker content trust
- Use Docker Bench for security
- Use Artifactory to manage Docker images
- Leverage Docker enterprise features for additional protection
- Writing a Dockerfile is always critical, build docker image which is slim and smart, not bloated
- Persist necessary data outside of a container using Volumes
- Use Docker compose to use as Infrastructure As Code and keep track using tags
- Role-based access control
- Do not add user credentials/keys/critical data to the image. Use it as a deployment variable
- Make use of docker caching, try pushing any "COPY . ." commands as close to the end of the Dockerfile as possible
- Use .dockerignore file
- Don't install debugging tools to reduce image size
- Always use resource limits with docker/containers
- Use swarm mode for small application
- Don't blindly trust downloads from the DockerHub! Verify them! See more at 'DockerHub Breach Can Have a Long Reach'
- Make Docker image with tuned kernel parameters
- Use alpine image

Learn more in the original article **'Why we love Docker and best practices for DevOps'**

Docker is in the news for two reasons: **Image retention limits** and **download throttling.**



- ***Image retention limits***
  Images stored on DockerHub with free accounts (which is very common for both open-source projects and automated builds) will be subject to a six-month image retention policy, after which their images will be deleted if they are inactive.

- ***Download throttling***
  Docker introduced a download rate limit of 100 pulls per six hours for anonymous users and 200 pulls per six hours for free accounts.

Solution

Two groups are potentially going to get affected by the new policies established by Docker - The open-source contributors who create Docker images and the DevOps enthusiasts who consume them.

The open-source group will have image retention issues for less used but important images. DevOps enthusiasts will have challenges using Docker Hub as a system of record, because images might disappear without warning, breaking builds, and anonymous or free builds can fail due to throttling of pull requests.

With the latest image retention and throttling policies by Docker, the free tier users and the open-source enthusiasts are going to get affected. These communities have no choice but to move to a highly sophisticated option, that is 'Artifactory'. Yes!

Why is Artifactory better than Docker Hub?

1. Artifactory caches images, so you are not affected by upstream removal.

2. Artifactory serves from the cache, so only 1 pull per image, preventing throttling.

3. Only a single DockerHub license is required for all developers and build machines in an organization.

4. Artifactory allows you to create multiple Docker registries per instance. You can make use of a local repository as a private Docker registry to share Docker images across the organization with fine-grained access control.

5. You can store and also retrieve any type of artifact, including secure docker images that your development teams produce, and having these artifacts stored in a central, managed location makes Artifactory an essential part of any software delivery lifecycle.

Learn more in the original article 'Mitigating DevOps Repository Risks'.

Containers have been getting a lot of attention, and special thanks to Docker for enabling firms to transform themselves digitally by building and shipping software at a faster rate than usual. Docker adoption has enhanced developer productivity with greater agility and higher efficiency. Kubernetes and Docker together are pretty powerful. Docker has had a huge impact on the toolsets and strategies for deploying and managing containers in general. Also, there isn't much competition to Docker containers other than cri-o or other open container options.
Let us wait and see how Docker moves as we advance.

Get Started with Artifactory as a Docker Registry