**Docker MasterClass for machine Learning and DataScience**

In this course:

We learn ins and outs of Docker, as well as Docker Swarm, Docker Compose, and using Docker with AWS. Also, your Docker path will cover the following steps:

* A Comprehensive Introduction to Docker
* Getting the Docker basics down
* Using Dockerhub
* Docker Challenges

------------------------------------------------------------------------------------

**What is Docker?**

is a tool that can package our applications or algorithms along with their dependencies in a virtual container that can be run on a linux server.

Helps in replicating environment, packages in which the code originally was built and dockerized

**What is a container?**

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one.

Multiple containers can run on the same machine and share same OS. Take less size and can handle more apps and require fewer VMs

**Container vs virtual Machine?**

Virtualization is not possible without the hypervisor. A hypervisor, or virtual machine monitor, is the software or firmware layer that enables multiple operating systems to run side-by-side, all with access to the same physical server resources. The hypervisor orchestrates and separates the available resources (computing power, memory, storage, etc.), aligning a portion to each virtual machine as needed.

**Virtualization** enables you to run multiple operating systems on the hardware of a single physical server, while **containerization** enables you to deploy multiple applications using the same operating system on a single virtual machine or server.

Virtual machines are great for supporting applications that require an operating system’s full functionality when you want to deploy multiple applications on a server, or when you have a wide variety of operating systems to manage. Containers are a better choice when your biggest priority is to minimize the number of servers you’re using for multiple applications.

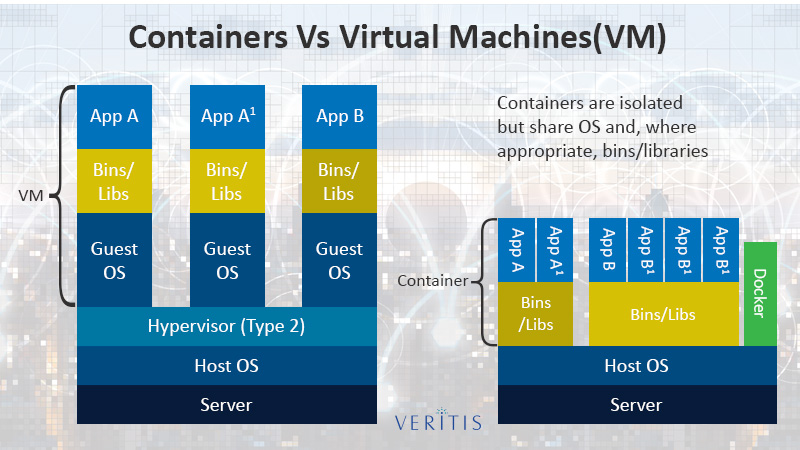
Your use case matters too. Containers are an excellent choice for tasks with a much shorter lifecycle. With their fast set up time, they are suitable for tasks that may only take a few hours. Virtual machines have a longer lifecycle than containers, and are best used for longer periods of time.

## **What are the advantages and disadvantages of virtual machines?**

Visually, each virtual machine image looks like a data folder. Each can be moved and copied as easily as you can move and copy other kinds of files. In this way, your team can centralize workloads and run several different operating systems without increasing overhead, a huge advantage over on-premises hardware. You can also update apps and the OS without affecting the end-user experience.

### However, virtual machines are not without their disadvantages:

* Since each VM includes an OS and a virtual copy of all the hardware the OS requires, VMs require significant RAM and CPU resources
* Due to the increase in virtual copies and required resources, the software development life cycle is more complex with VMs
* Moving VMs between public clouds, private clouds and traditional data centers can be challenging



---------------------------------------------------------------------------------------------------------------------

**Docker In Depth:**

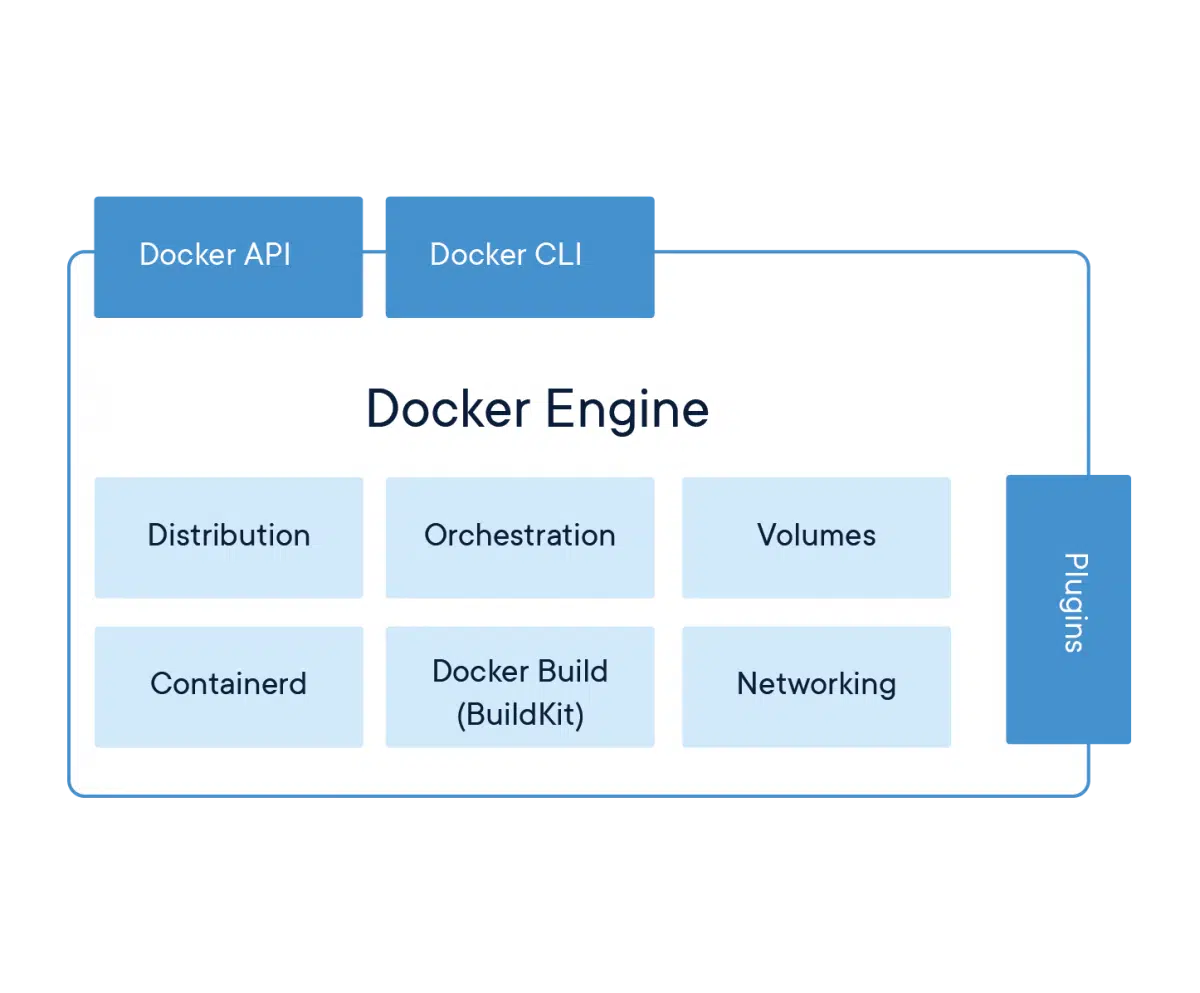
## **Package Software into Standardized Units for Development, Shipment and Deployment**

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, stand-alone, executable package of software that includes everything needed to run an application: **code, runtime, system tools, system libraries and settings**.

Container images become containers at runtime and in the case of Docker containers – images become containers when they run on [Docker Engine](https://www.docker.com/products/container-runtime). Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Docker containers that run on Docker Engine:

1. Standard: Docker created the industry standard for containers, so they could be portable anywhere
2. Lightweight: Containers share the machine’s OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs
3. Secure: Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry



-----------------------------------------------------------------------------------------------------------------

# Docker overview

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

## The Docker platform

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allows you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. You can easily share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Docker provides tooling and a platform to manage the lifecycle of your containers:

* Develop your application and its supporting components using containers.
* The container becomes the unit for distributing and testing your application.
* When you’re ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

## What can I use Docker for?

**Fast, consistent delivery of your applications**

Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide your applications and services. Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

Consider the following example scenario:

* Your developers write code locally and share their work with their colleagues using Docker containers.
* They use Docker to push their applications into a test environment and execute automated and manual tests.
* When developers find bugs, they can fix them in the development environment and redeploy them to the test environment for testing and validation.
* When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.

**Responsive deployment and scaling**

Docker’s container-based platform allows for highly portable workloads. Docker containers can run on a developer’s local laptop, on physical or virtual machines in a data center, on cloud providers, or in a mixture of environments.

Docker’s portability and lightweight nature also make it easy to dynamically manage workloads, scaling up or tearing down applications and services as business needs dictate, in near real time.

**Running more workloads on the same hardware**

Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines, so you can use more of your compute capacity to achieve your business goals. Docker is perfect for high density environments and for small and medium deployments where you need to do more with fewer resources.

-----------------------------------------------------------------------------------------------------------------

## Docker architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

### The Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

### The Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

### Docker Desktop

Docker Desktop is an easy-to-install application for your Mac or Windows environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

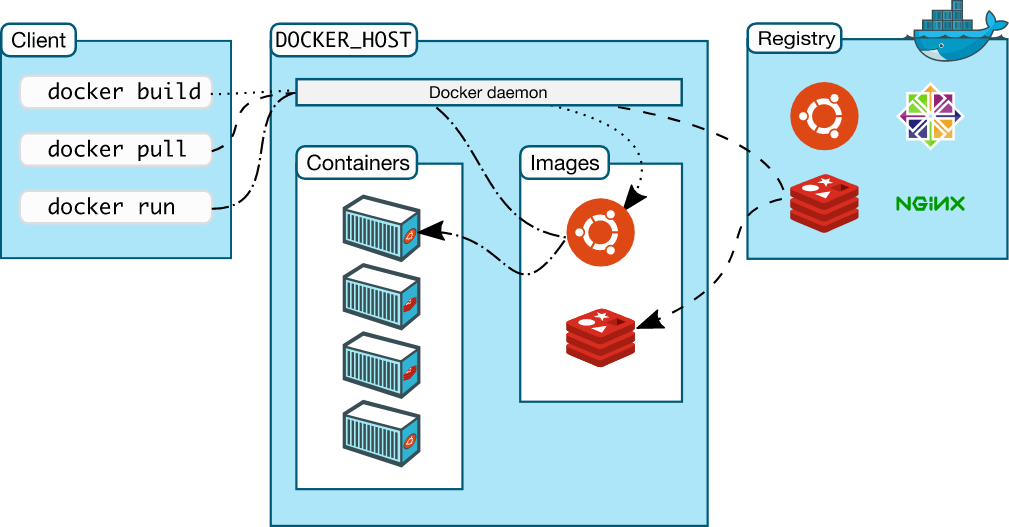
### Docker registries

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

### Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.



#### Images

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

#### Containers

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container’s network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

##### Example docker run command

The following command runs an ubuntu container, attaches interactively to your local command-line session, and runs /bin/bash.

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

When you run this command, the following happens (assuming you are using the default registry configuration):

1. If you do not have the ubuntu image locally, Docker pulls it from your configured registry, as though you had run docker pull ubuntu manually.
2. Docker creates a new container, as though you had run a docker container create command manually.
3. Docker allocates a read-write filesystem to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.
4. Docker creates a network interface to connect the container to the default network, since you did not specify any networking options. This includes assigning an IP address to the container. By default, containers can connect to external networks using the host machine’s network connection.
5. Docker starts the container and executes /bin/bash. Because the container is running interactively and attached to your terminal (due to the -i and -t flags), you can provide input using your keyboard while the output is logged to your terminal.
6. When you type exit to terminate the /bin/bash command, the container stops but is not removed. You can start it again or remove it.

-----------------------------------------------------------------------------------------------------------------

**Installing Docker for Linux**

The Docker desktop dashboard currently isn't native for Linux, but you can set up Docker on a Linux distribution. To do so, you can use the following option:

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

[https://docs.docker.com/engine/install/#other-linux-distributions](https://docs.docker.com/engine/install/" \l "other-linux-distributions)

Also, you can see example steps here for an Ubuntu setup: <https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04>

A VM is also an option for the desktop dashboard with Windows or Mac but you can use the terminal on Linux to replicate commands

**How Docker Is Useful For Data Science**

We all know that the domain of Data Science is developing and advancing rapidly, and we are always picking up new tools to use to learn and assist us with our careers. Sometimes we might wonder whether the software, library, or tool will be around for the foreseeable future and one thing is certain, and that is that Docker and containers are here to stay.

**Examples of Docker benefits include:**

* Uniform requirements
* Easily shared across systems and teams
* Highly scalable
* Resource optimization
* Isolation
* Dependency management

=======================================================

**Build a simple Docker App:**

Build a simple flask file as below:

**# -\*- coding: utf-8 -\*-**

**from flask import Flask**

**app = Flask(\_\_name\_\_)**

**@app.route('/')**

**def example():**

**return "A simple Docker App"**

**if \_\_name\_\_ == "\_\_main\_\_":**

**app.run(host = '0.0.0.0', port=int('5000'), debug=True)**

Build a simple DockerFile as below:

**FROM python:alpine3.8**

**COPY . /app**

**WORKDIR /app**

**RUN pip install -r requirements.txt**

**EXPOSE 5000**

**CMD python ./simple\_Docker\_app.py**

Create a simple requirements.txt file as below:

**flask == 2.1.2**

**Then go to root directory,**

**run docker build -t testimage .**

**Then run,**

**docker images –all**

**Then run,**

**docker run –name testcontainer -p 5000:5000 testimage**

**An example has been shown below:**

**(base) C:\Users\sadiq>conda activate testDocker**

**(testDocker) C:\Users\sadiq\OneDrive\Documents\Docker MasterClass for ML and DS>docker build -t testimage .**

**[+] Building 2.7s (9/9) FINISHED**

**=> [internal] load build definition from Dockerfile 0.0s**

**=> => transferring dockerfile: 32B 0.0s**

**=> [internal] load .dockerignore 0.0s**

**=> => transferring context: 2B 0.0s**

**=> [internal] load metadata for docker.io/library/python:alpine3.8 2.6s**

**=> [internal] load build context 0.0s**

**=> => transferring context: 330B 0.0s**

**=> [1/4] FROM docker.io/library/python:alpine3.8@sha256:3491d1abd29b3f87ca5cb1afd34bc696855a2403df1ff854da55cb67 0.0s**

**=> CACHED [2/4] COPY . /app 0.0s**

**=> CACHED [3/4] WORKDIR /app 0.0s**

**=> CACHED [4/4] RUN pip install -r requirements.txt 0.0s**

**=> exporting to image 0.0s**

**=> => exporting layers 0.0s**

**=> => writing image sha256:5d7e02d7380be8864acd2d2f4b60079858a10264d72eeae72e0f38a5e483d65f 0.0s**

**=> => naming to docker.io/library/testimage 0.0s**

**(testDocker) C:\Users\sadiq\OneDrive\Documents\Docker MasterClass for ML and DS>docker images --all**

**REPOSITORY TAG IMAGE ID CREATED SIZE**

**testimage latest 5d7e02d7380b 10 minutes ago 90.3MB**

**(testDocker) C:\Users\sadiq\OneDrive\Documents\Docker MasterClass for ML and DS>docker run --name testcontainer -p 5000:5000 testimage**

**\* Serving Flask app 'simple\_Docker\_app' (lazy loading)**

**\* Environment: production**

**WARNING: This is a development server. Do not use it in a production deployment.**

**Use a production WSGI server instead.**

**\* Debug mode: on**

**\* Running on all addresses (0.0.0.0)**

**WARNING: This is a development server. Do not use it in a production deployment.**

**\* Running on http://127.0.0.1:5000**

**\* Running on http://172.17.0.2:5000 (Press CTRL+C to quit)**

**\* Restarting with stat**

**\* Debugger is active!**

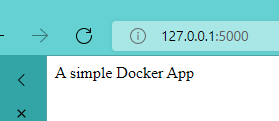
**\* Debugger PIN: 920-439-482**

**172.17.0.1 - - [10/May/2022 04:01:14] "GET / HTTP/1.1" 200 -**

**172.17.0.1 - - [10/May/2022 04:04:46] "GET / HTTP/1.1" 200 -**

**===============================================**

**One weblink with open port is generated, you may verify it on browser.**

****

# **Docker Container Security 101: Risks and 33 Best Practices**

**Containers, along with orchestrators such as Kubernetes, have ushered in a new era of application development methodology, enabling microservices architectures as well as continuous development and delivery. Docker is by far the most dominant container runtime engine, with a 91% penetration according to our latest**[**State of the Container and Kubernetes Security Report**](https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-2021-overview)**.**

Containerization has many benefits and as a result has seen wide adoption. According to Gartner, by 2020, more than 50% of global organizations will be running containerized applications in production. However, building apps using Docker containers also introduces new security challenges and risks. A single compromised Docker container can threaten all other containers as well as the underlying host, underscoring the importance of securing Docker.

Securing Docker can be loosely categorized into two areas: securing and hardening the host so that a container breach doesn’t also lead to host breach, and securing Docker containers. We have briefly covered host security in a [previous blog article](https://cloud.redhat.com/blog/hardening-docker-containers-images-and-host-security-toolkit). This article focuses on container security by highlighting Docker container security risks and challenges as well as providing best practices for hardening your environment during the build and deploy phases and protecting your Docker containers during runtime.

We also share best practices for securing Kubernetes, given its massive adoption and critical role in orchestrating containers. Finally, we provide you with 11 key security questions your container security platform should be able to answer, giving you the insights and protection you need to run containers and Kubernetes securely in production.

## 8 container security challenges you must address for Docker

Companies have long deployed applications on virtual machines (VMs) or bare metal servers. Security for that infrastructure involved securing your application and the host it’s running on and then protecting the application as it runs. Containerization introduces several new challenges that must be addressed.

1. Containers enable microservices, which increases data traffic and network and access control complexity.
2. Containers rely on a base image, and knowing whether the image comes from a secure or insecure source can be challenging. Images can also contain vulnerabilities that can spread to all containers that use the vulnerable image.
3. Containers have short life spans, so monitoring them, especially during runtime, can be extremely difficult. Another security risk arises from a lack of visibility into an ever-changing container environment.
4. Containers, unlike VMs, aren’t necessarily isolated from one another. A single compromised container can lead to other containers being compromised.
5. Containerized environments have many more components than traditional VMs, including the Kubernetes orchestrator that poses [its own set of security challenges](https://www.stackrox.io/blog/kubernetes-security-101/). Can you tell which deployments or clusters are affected by a high-severity vulnerability? Are any exposed to the Internet? What’s the blast radius if a given vulnerability is exploited? Is the container running in production or a dev/test environment?
6. Container configuration is yet another area that poses security risks. Are containers running with heightened privileges when they shouldn’t? Are images launching unnecessary services that increase the attack surface? Are secrets stored in images?
7. As one of the biggest security drivers, compliance can be a particular challenge given the fast-moving nature of container environments. Many of the traditional components that helped demonstrate compliance, such as firewall rules, take a very different form in a Docker environment.
8. Finally, existing server workload security solutions are ill-equipped to address container security challenges and risks.

## **26 Docker security best practices**

What follows is a list of best practices derived from industry standards and StackRox customers for securely configuring your Docker containers and images.

1. Always use the most up to date version of Docker. The [runC vulnerability](https://www.stackrox.io/blog/the-runc-vulnerability-a-deep-dive-on-protecting-yourself/) from earlier this year, for example, was quickly patched soon after its discovery with the release of Docker version [18.09.2](https://github.com/docker/docker-ce/releases/tag/v18.09.2).
2. Allow only trusted users control of the Docker daemon by making sure only trusted users are members of Docker group. Check out [this article](_blank) for more information about decreasing your Docker daemon attack surface.
3. Make sure you have rules in place that give you an audit trail for:
   * Docker daemon
   * Docker files and directories:
     + /var/lib/docker
     + /etc/docker
     + Docker.service
     + Docker.socket
     + /etc/default/docker
     + /etc/docker/daemon.json
     + /etc/sysconfig/docker
     + /usr/bin/containerd
     + /usr/sbin/runc

Check out [this article](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing) for more details

1. Secure all Docker files and directories (see 4.2 above) by ensuring they are owned by the appropriate user (usually the root user) and their file permissions are set to a restrictive value (see the [CIS benchmarks](https://docs.docker.com/v17.09/compliance/cis/) section on Docker daemon configuration files).
2. Use registries that have a valid registry certificate or ones that use TLS to minimize the risk of traffic interception.
3. If you are using containers without an explicit container user defined in the image, you should enable user namespace support, which will allow you to re-map container user to host user.
4. Disallow containers from acquiring new privileges. By default, containers are allowed to acquire new privileges so this configuration must be explicitly set. Another step you can take to minimize a privilege escalation attack is to remove the setuid and setgid permissions in the images.
5. As a best practice, run your containers as a non-root user (UID not 0). By default, containers run with root privileges as the root user inside the container.
6. Use only trusted base images when building your containers. This tip might seem like an obvious one, but third-party registries often don’t have any governance policies for the images stored in them. It’s important to know which images are available for use on the Docker host, understand their provenance, and review the content in them. You should also enable Content trust for Docker for image verification and install only verified packages into images.
7. Use minimal base images that don’t include unnecessary software packages that could lead to a larger attack surface. Having fewer components in your container reduces the number of available attack vectors, and a minimal image also yields better performance because there are fewer bytes on disk and less network traffic for images being copied. BusyBox and Apline are two options for building minimal base images.
8. Implement a strong governance policy that enforces frequent image scanning. Stale images or images that haven’t been scanned recently should be rejected or rescanned before moving to build stage.
9. Build a workflow that regularly identifies and removes stale or unused images and containers from the host.
10. Don’t store secrets in images/Dockerfiles. By default, you’re allowed to store secrets in Dockerfiles, but storing secrets in an image gives any user of that image access to the secret. When a secret is required, use a [secrets management tool](https://blog.docker.com/2017/02/docker-secrets-management/).
11. When running containers, remove all capabilities not required for the container to function as needed. You can use Docker’s CAP DROP capability to drop a specific container’s capabilities (also called Linux capability), and use CAP ADD to add only those capabilities required for the proper functioning of the container.
12. Don’t run containers with --privileged flag, as this type of container will have most of the capabilities available to the underlying host. This flag also overwrites any rules you set using CAP DROP or CAP ADD.
13. Don’t mount sensitive host system directories on containers, especially in writable mode that could expose them to being changed maliciously in a way that could [lead to host compromise](_blank).
14. Don’t run sshd within containers. By default, the ssh daemon will not be running in a container, and you shouldn’t install the ssh daemon to simplify security management of the SSH server.
15. Don’t map any ports below 1024 within a container as they are considered privileged because they transmit sensitive data. By default, Docker maps container ports to one that’s within the 49153–65525 range, but it allows the container to be mapped to a privileged port. As a general rule of thumb, ensure only needed ports are open on the container.
16. Don’t share the host’s network namespace, process namespace, IPC namespace, user namespace, or UTS namespace, unless necessary, to ensure proper isolation between Docker containers and the underlying host.
17. Specify the amount of memory and CPU needed for a container to operate as designed instead of relying on an arbitrary amount. By default, Docker containers share their resources equally with no limits.
18. Set the container’s root filesystem to read-only. Once running, containers don’t need changes to the root filesystem. Any changes made to the root filesystem will likely be for a malicious objective. To preserve the immutable nature of containers — where new containers don’t get patched but rather recreated from a new image — you should not make the root filesystem writable.
19. Impose PID limits. One of the advantages of containers is tight process identifier (PID) control. Each process in the kernel carries a unique PID, and containers leverage Linux PID namespace to provide a separate view of the PID hierarchy for each container. Putting limits on PIDs effectively limits the number of processes running in each container. Limiting the number of processes in the container prevents excessive spawning of new processes and potential malicious lateral movement. Imposing PID limits also prevents fork bombs (processes that continually replicate themselves) and anomalous processes. Mostly, the benefit here is if your service always runs a specific number of processes, then setting the PID limit to that exact number mitigates many malicious actions, including reverse shells and remote code injection — really, anything that requires spawning a new process.
20. Don’t configure your mount propagation rules as shared. Sharing mount propagation means that any changes made to the mount will propagate to all instances of that mount. Instead set the mount propagation in slave or private mode so that a necessary change made to a volume isn’t shared with (or propagated to) containers that don’t require that change.
21. Don’t use docker exec command with privileged or user=root option, since this setting could give the container extended Linux capabilities
22. Don’t use the default bridge “docker0.” Using the default bridge leaves you open to ARP spoofing and MAC flooding attacks. Instead containers should be on a user-defined network and not the default “docker0” bridge.
23. Don’t mount Docker socket inside containers, since this approach would allow a process within the container to execute commands that give it full control of the host.

## 7 Kubernetes security best practices

As the de facto standard for container orchestration, Kubernetes plays a pivotal role in ensuring your applications are secure. To effectively secure containerized applications, you must leverage the contextual information from Kubernetes as well as its native policy enforcement capabilities. For example, Kubernetes has several built-in security features that make it easier to operationalize full life cycle container security, including Kubernetes RBAC, Network Policies, and Admission Controllers. Tap into the power of these inherent control capabilities in Kubernetes to protect your containerized environments.

Below are some Kubernetes security best practices that help operationalize full life cycle container security.

1. For RBAC, specify your Roles and ClusterRoles to specific users or groups of users instead of granting cluster-admin privileges to any user or groups of users.
2. Avoid duplication of permissions when using Kubernetes RBAC, as doing so could create operational issues.
3. Remove unused or inactive RBAC roles so that you can focus your attention on the active roles when troubleshooting or investigating security incidents.
4. Use Kubernetes network policies to isolate your pods and explicitly allow only the communication paths required for the application to function. Otherwise you’re exposing yourself to both lateral and north-south threats.
5. If your pods need Internet access (either ingress or egress), create the appropriate network policy that enforces the right network segmentation/firewalling rule, then create a label that’s targeted by the said network policy, and lastly associate your pods to that label.
6. Use the PodSecurityPolicy admission controller to ensure proper governance policies are being enforced. The PodSecurityPolicy controller can prevent containers from running as root or make sure a container’s root filesystem is mounted read-only (these recommendations should sound familiar, given they’re both on the previous list of Docker measures to take).
7. Use the Kubernetes admission controller to enforce image registry governance policies such that all images taken from untrusted registries are automatically denied.

## Final thoughts — make sure you can answer these 11 security questions about your Docker container environment

To help you quickly assess your security posture, we’ve compiled a list of questions your security, DevSecOps, or DevOps teams should readily be able to answer if your cloud-native stack has been architected with appropriate security measures.

1. How many images are on a host where the last scan date exceeds 60 days?
2. How many images/containers have a high-severity vulnerability?
3. Which deployments are impacted by these high-severity vulnerable containers?
4. Are there any containers in the impacted deployments that have secrets stored in them?
5. Are any of the vulnerable containers running as root or with privileged flag?
6. Are any of the vulnerable containers in a Pod that doesn’t have a network policy associated to it (meaning it allows all communication)?
7. Are any containers running in production impacted by this vulnerability?
8. Where are the images we’re using coming from?
9. How are we blocking images that are being pulled from untrusted registries?
10. Are we able to see which processes are executing during container runtime?
11. Which clusters, namespaces, and nodes are non-compliant with CIS benchmarks for Docker and Kubernetes?

Follow the best practices compiled in these lists and you’ll have taken the most important steps to successfully hardening your Docker and Kubernetes environments and protecting your critical business applications.

# Hardening Docker containers, images, and host - security toolkit

## Introduction

Container technology has radically changed the way that applications are being developed and deployed. Notably, containers dramatically ease dependency management, so shipping new features or code is faster than ever before. While Docker containers and Kubernetes are great for DevOps, they also present new security challenges that both security practitioners and developers must understand and address with diligence. Docker’s team of security experts has built some valuable security features into the Docker platform over the last several years. This blog post offers up valuable tips and practical suggestions on how containers and hosts can be hardened as a first step toward a more secure container environment. You should also check out our blog post on [container security best practices](https://www.stackrox.com/post/2018/12/6-container-security-best-practices-you-should-be-following/) you should be following.

### A BRIEF LOOK AT CONTAINERS FROM A SECURITY PERSPECTIVE

In essence, Docker containers are a wrapper around Linux control groups (cgroups) and namespaces. Cgroups are used in the Linux kernel for monitoring and restricting resources among a group of processes. Namespaces determine what a process can see. For example, the PID namespace restricts which processes can be seen within a container.

Each container running on a host shares a common underlying kernel. Containers are isolated from one another, which – from a security standpoint – is advantageous. However, if the host OS is compromised, all containers running on it are at risk. Similarly, if a container is using a vulnerable library, it could be exploited to gain access to the underlying host.

## Securing the host OS

The underlying host OS needs to be secured in order to prevent container breaches from affecting the host. For this, Linux provides several out-of-the-box security modules. Some of the popular ones are **SELinux**, **AppArmor** and **seccomp**. One can also develop custom security modules using **Linux Security Modules** (LSMs).

### SELINUX

SELinux is a type of Mandatory Access Control (MAC) security module based on type enforcement. Type enforcement revolves around defining a type and assigning privileges to those types. Here’s a simple example of using an SELinux policy:

policy\_module(localpolicy, 1.0)

gen\_require('

type user\_t;

type var\_log\_t;

)

allow user\_t var\_log\_t:dir { getattr search open read };

This allows any Linux process to execute file operations (such as getattr, search, open, read) for which user\_t and var\_log\_t are applied.

Even though SELinux is comprehensive, you can see the policy language is pretty hard to get right. Often users will end up creating different SELinux policies over time to address different scenarios.

### APPARMOR

AppArmor is another MAC solution. It is based on file system paths rather than defining types. Users can specify a file path to a binary and the permissions they have. Here’s a simple example of confining nginx:

#include <tunables/global>

/usr/sbin/nginx {

#include <abstractions/apache2-common>

#include <abstractions/base>

#include <abstractions/nis>

capability dac\_override,

capability dac\_read\_search,

capability net\_bind\_service,

capability setgid,

capability setuid,

/data/www/safe/\* r,

deny /data/www/unsafe/\* r,

/etc/group r,

/etc/nginx/conf.d/ r,

/etc/nginx/mime.types r,

/etc/nginx/nginx.conf r,

/etc/nsswitch.conf r,

/etc/passwd r,

/etc/ssl/openssl.cnf r,

/run/nginx.pid rw,

/usr/sbin/nginx mr,

/var/log/nginx/access.log w,

/var/log/nginx/error.log w,

}

This AppArmor profile blocks reading from unsafe directories. It may not be suitable for all situations; users may have to customize it. AppArmor is good for restricting application access. However, it requires a learning curve to be able to write good enforcement profiles.

### SECCOMP

Seccomp (short for ‘Secure Computing’) is another security module included in many Linux distributions that allows users to restrict system calls. Users can specify custom actions to be taken when a certain system call is executed. The actions are allow, kill, err, and trap. Seccomp can be used to sandbox applications that handle untrusted user inputs to a subset of system calls.

The first step in using seccomp is to determine all the system calls an application makes when it runs. This can be a difficult and error-prone exercise that should be conducted when the application is written. Users can use tools like audit to profile all the system calls that it makes by exercising it in different ways.

Seccomp policies are defined using JSON files. A sample seccomp policy looks like this,

{

"defaultAction": "SCMP\_ACT\_ALLOW",

"syscalls": [

{

"name": "mkdir",

"action": "SCMP\_ACT\_ERRNO"

},

{

"name": "chown",

"action": "SCMP\_ACT\_ERRNO"

}

]

}

This policy causes an error to be returned when mkdir or chown are executed.

The drawback with seccomp is that the profile has to be applied during the launch of the application. The granularity of restricting system calls is too narrow and requires extensive working knowledge of Linux to come up with good profiles.

### CAPABILITIES

Linux capabilities are groups of permissions that can be given to child processes. Child processes cannot acquire newer capabilities. The idea behind capabilities is that no process should have all privileges, but instead, have only enough privileges to perform their intended service. By bootstrapping processes with limited privileges, we can “contain” the damage that can occur if they are ever compromised.

Some capabilities give excessive privileges to processes such as:

* **SYS\_ADMIN**: This gives processes many privileges, some of which are given only to the root user.
* **SETUID**: Many of the Linux distributions ship binaries that run with the setuid bit set to give root privileges by default. Instead of setuid bit, they can be replaced with capabilities to provide more granular privileges.

These options work well on the host OS, but it can be challenging to adapt them for containers. Let’s look at some of the relevant attack surfaces and practical ways to secure the Docker container runtime.

## Container Runtime Security Practices

There are various factors to consider when adopting Docker containers for production. When it comes to running Docker container securely, users can follow these recommendations.

### UNIX SOCKET (/VAR/RUN/DOCKER.SOCK)

By default, the Docker client communicates with the Docker daemon using the unix socket. This socket can also be mounted by any other container unless proper permissions are in place. Once mounted, it is very easy to spin up any container, create new images, or shut down existing containers.

**Solution**: Set up appropriate SELinux/AppArmor profiles to limit containers mounting this socket.

### VOLUME MOUNTS

Docker allows mounting to sensitive host directories. Also, the contents of the host file system can be changed directly from the container. For application containers with direct Internet exposure, it is important to be extra careful when mounting sensitive host directories (/etc/, /usr/). Any breach can lead to damaging data loss.

**Solution:** Mount host-sensitive directories as read-only.

### PRIVILEGED CONTAINERS

Privileged containers can do almost anything a host can do. It runs with all capabilities.

**Solution:** Use capabilities to grant fine-grained privileges instead.

### SSH WITHIN CONTAINER

Running ssh service within containers makes managing ssh keys/ access policies difficult. This should be avoided if possible.

**Solution:**

* Do not run ssh services inside a container.
* Instead, run ssh on the host and use docker exec or docker attach to interact with the container.

### BINDING PRIVILEGED PORTS

By default, Docker allows binding privileged ports **(<1024)** to a container. Normal users cannot access these ports. In many cases, mapping **http port 80** and **https port 443** is necessary for running servers in a container.

**Solution:**

* List all containers and their port mappings using the code below to ensure that the container’s ports are not mapped to host ports below port 1024.
* docker ps --quiet | xargs docker inspect --format ': Ports='

### EXPOSING PORTS

Ports not necessary for the service must not be exposed.

**Solution:** List all the containers and their exposed ports using the following:

* docker ps --quiet | xargs docker inspect --format ': Ports='
* Ensure that there are no unnecessary ports exposed.

### RUNNING WITHOUT DEFAULT APPARMOR/ SELINUX OR SECCOMP

Docker runs containers with default AppArmor/SELinux and seccomp profiles. They can be disabled with the --unconfined option.

**Solution:** Do not disable the default profiles that Docker supplies.

### SHARING HOST NAMESPACES

Sharing namespaces have dangerous consequences if not managed properly. Containers can be started with -pid to connect with the host PID namespace or --net to share its network namespace. These allow containers to see and kill PIDs running on the host or even connect to privileged ports.

**Solution:** Avoid sharing host namespaces with containers.

### ENABLING TLS

If the Docker daemon is running on a TCP endpoint, it is advised to run with TLS enabled.

**Solution:** Docker offers a [helpful guide on enabling TLS with Docker](https://docs.docker.com/engine/security/https/).

### DO NOT SET MOUNT PROPAGATION MODE TO SHARED

Mount propagation mode allows mounting volumes in shared, slave or private mode on a container. Do not use shared mount propagation mode until needed.

A shared mount is replicated at all mounts and the changes made at any mount point are propagated to all mounts. Mounting a volume in shared mode does not restrict any other container to mount and make changes to that volume. This might be catastrophic if the mounted volume is sensitive to changes. Do not set mount propagation mode to shared until needed.

**Solution:** Run the following command to list the propagation mode for mounted volumes:

* docker ps --quiet --all | xargs docker inspect --format ': Propagation= '
* Ensure that the mode is not set to shared unless needed.
* Do not start a container with the following invocation:
* docker run --volume=/hostPath:/containerPath:shared <image> <command>

### RESTRICT A CONTAINER FROM ACQUIRING NEW PRIVILEGES

A process can set the no\_new\_priv bit in the kernel. It persists across fork, clone and execve. The no\_new\_priv bit ensures that the process or its children processes do not gain any additional privileges via setuid or sgid bits.

**Solution:**

List the security options for all the containers using the following command:

* docker ps --quiet --all | xargs docker inspect --format ': SecurityOpt='
* The security options should list no\_new\_privileges as one of them.
* One can start a container with no\_new\_privileges as below:
* docker run <run-options> --security-opt=no-new-privileges <image> <cmd>

## Conclusion

Naturally, new technologies presents new security challenges to organizations that choose to deploy it. As has been the case with the disruptive infrastructure technologies that preceded containers, the first step toward establishing a stronger security posture is to reduce the overall attack surface by hardening it. While the practices presented above are effective in making containers and hosts far less susceptible to exploits, the major container security challenges lie within the runtime phase. This is where it is critical for enterprise organizations to employ a dedicated container security platform.

## References

<https://www.cisecurity.org/benchmark/docker/>

**Docker Hub**

First, in order to use Docker Hub you can sign up at the following link. Signing up is completely free:

<https://hub.docker.com/>

Once you set up a repository you can find the related commands to push an image to the repository. You can push it using the Docker push command:

Docker push username/reponame:tag

To pull an image it’s a simple process via Docker Hub. You can select the repository or image that you would like to pull and find the docker pull command. In our lecture we used couchbase as an example with the command:

Docker pull couchbase

We can then use the Docker run command to run the image and build a container.

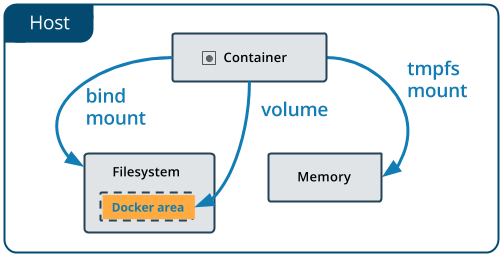
Always remember to use only trusted, verified or certified images when possible.

**Docker Volumes:**

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While [bind mounts](https://docs.docker.com/storage/bind-mounts/) are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

* Volumes are easier to back up or migrate than bind mounts.
* You can manage volumes using Docker CLI commands or the Docker API.
* Volumes work on both Linux and Windows containers.
* Volumes can be more safely shared among multiple containers.
* Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
* New volumes can have their content pre-populated by a container.
* Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

In addition, volumes are often a better choice than persisting data in a container’s writable layer, because a volume does not increase the size of the containers using it, and the volume’s contents exist outside the lifecycle of a given container.

****

If your container generates non-persistent state data, consider using a [tmpfs mount](https://docs.docker.com/storage/tmpfs/) to avoid storing the data anywhere permanently, and to increase the container’s performance by avoiding writing into the container’s writable layer.

Volumes use rprivate bind propagation, and bind propagation is not configurable for volumes.

## Choose the -v or --mount flag

In general, --mount is more explicit and verbose. The biggest difference is that the -v syntax combines all the options together in one field, while the --mount syntax separates them. Here is a comparison of the syntax for each flag.

If you need to specify volume driver options, you must use --mount.

* **-v or --volume**: Consists of three fields, separated by colon characters (:). The fields must be in the correct order, and the meaning of each field is not immediately obvious.
  + In the case of named volumes, the first field is the name of the volume, and is unique on a given host machine. For anonymous volumes, the first field is omitted.
  + The second field is the path where the file or directory are mounted in the container.
  + The third field is optional, and is a comma-separated list of options, such as ro. These options are discussed below.
* **--mount**: Consists of multiple key-value pairs, separated by commas and each consisting of a <key>=<value> tuple. The --mount syntax is more verbose than -v or --volume, but the order of the keys is not significant, and the value of the flag is easier to understand.
  + The type of the mount, which can be [bind](https://docs.docker.com/storage/bind-mounts/), volume, or [tmpfs](https://docs.docker.com/storage/tmpfs/). This topic discusses volumes, so the type is always volume.
  + The source of the mount. For named volumes, this is the name of the volume. For anonymous volumes, this field is omitted. May be specified as source or src.
  + The destination takes as its value the path where the file or directory is mounted in the container. May be specified as destination, dst, or target.
  + The readonly option, if present, causes the bind mount to be [mounted into the container as read-only](https://docs.docker.com/storage/volumes/" \l "use-a-read-only-volume). May be specified as readonly or ro.
  + The volume-opt option, which can be specified more than once, takes a key-value pair consisting of the option name and its value.

Escape values from outer CSV parser

If your volume driver accepts a comma-separated list as an option, you must escape the value from the outer CSV parser. To escape a volume-opt, surround it with double quotes (") and surround the entire mount parameter with single quotes (').

For example, the local driver accepts mount options as a comma-separated list in the o parameter. This example shows the correct way to escape the list.

$ docker service create \

--mount 'type=volume,src=<VOLUME-NAME>,dst=<CONTAINER-PATH>,volume-driver=local,volume-opt=type=nfs,volume-opt=device=<nfs-server>:<nfs-path>,"volume-opt=o=addr=<nfs-address>,vers=4,soft,timeo=180,bg,tcp,rw"'

--name myservice \

<IMAGE>

The examples below show both the --mount and -v syntax where possible, and --mount is presented first.

### Differences between -v and --mount behavior

As opposed to bind mounts, all options for volumes are available for both --mount and -v flags.

When using volumes with services, only --mount is supported.

## Create and manage volumes

Unlike a bind mount, you can create and manage volumes outside the scope of any container.

**Create a volume**:

$ docker volume create my-vol

**List volumes**:

$ docker volume ls

local my-vol

**Inspect a volume**:

$ docker volume inspect my-vol

[

{

"Driver": "local",

"Labels": {},

"Mountpoint": "/var/lib/docker/volumes/my-vol/\_data",

"Name": "my-vol",

"Options": {},

"Scope": "local"

}

]

**Remove a volume**:

$ docker volume rm my-vol

## Start a container with a volume

If you start a container with a volume that does not yet exist, Docker creates the volume for you. The following example mounts the volume myvol2 into /app/ in the container.

The -v and --mount examples below produce the same result. You can’t run them both unless you remove the devtest container and the myvol2 volume after running the first one.

* --mount
* -v

$ docker run -d \

--name devtest \

--mount source=myvol2,target=/app \

nginx:latest

Use docker inspect devtest to verify that the volume was created and mounted correctly. Look for the Mounts section:

"Mounts": [

{

"Type": "volume",

"Name": "myvol2",

"Source": "/var/lib/docker/volumes/myvol2/\_data",

"Destination": "/app",

"Driver": "local",

"Mode": "",

"RW": true,

"Propagation": ""

}

],

This shows that the mount is a volume, it shows the correct source and destination, and that the mount is read-write.

Stop the container and remove the volume. Note volume removal is a separate step.

$ docker container stop devtest

$ docker container rm devtest

$ docker volume rm myvol2

## Use a volume with docker-compose

A single docker compose service with a volume looks like this:

version: "3.9"

services:

frontend:

image: node:lts

volumes:

- myapp:/home/node/app

volumes:

myapp:

On the first invocation of docker-compose up the volume will be created. The same volume will be reused on following invocations.

A volume may be created directly outside of compose with docker volume create and then referenced inside docker-compose.yml as follows:

version: "3.9"

services:

frontend:

image: node:lts

volumes:

- myapp:/home/node/app

volumes:

myapp:

external: true

For more information about using volumes with compose see [the compose reference](https://docs.docker.com/compose/compose-file/compose-file-v3/" \l "volume-configuration-reference).

### Start a service with volumes

When you start a service and define a volume, each service container uses its own local volume. None of the containers can share this data if you use the local volume driver, but some volume drivers do support shared storage. Docker for AWS and Docker for Azure both support persistent storage using the Cloudstor plugin.

The following example starts a nginx service with four replicas, each of which uses a local volume called myvol2.

$ docker service create -d \

--replicas=4 \

--name devtest-service \

--mount source=myvol2,target=/app \

nginx:latest

Use docker service ps devtest-service to verify that the service is running:

$ docker service ps devtest-service

Removing the service does not remove any volumes created by the service. Volume removal is a separate step.

#### Syntax differences for services

The docker service create command does not support the -v or --volume flag. When mounting a volume into a service’s containers, you must use the --mount flag.

### Populate a volume using a container

If you start a container which creates a new volume, as above, and the container has files or directories in the directory to be mounted (such as /app/ above), the directory’s contents are copied into the volume. The container then mounts and uses the volume, and other containers which use the volume also have access to the pre-populated content.

To illustrate this, this example starts an nginx container and populates the new volume nginx-vol with the contents of the container’s /usr/share/nginx/html directory, which is where Nginx stores its default HTML content.

The --mount and -v examples have the same end result.

* --mount
* -v

$ docker run -d \

--name=nginxtest \

--mount source=nginx-vol,destination=/usr/share/nginx/html \

nginx:latest

After running either of these examples, run the following commands to clean up the containers and volumes. Note volume removal is a separate step.

$ docker container stop nginxtest

$ docker container rm nginxtest

$ docker volume rm nginx-vol

## Use a read-only volume

For some development applications, the container needs to write into the bind mount so that changes are propagated back to the Docker host. At other times, the container only needs read access to the data. Remember that multiple containers can mount the same volume, and it can be mounted read-write for some of them and read-only for others, at the same time.

This example modifies the one above but mounts the directory as a read-only volume, by adding ro to the (empty by default) list of options, after the mount point within the container. Where multiple options are present, separate them by commas.

The --mount and -v examples have the same result.

* --mount
* -v

$ docker run -d \

--name=nginxtest \

--mount source=nginx-vol,destination=/usr/share/nginx/html,readonly \

nginx:latest

Use docker inspect nginxtest to verify that the readonly mount was created correctly. Look for the Mounts section:

"Mounts": [

{

"Type": "volume",

"Name": "nginx-vol",

"Source": "/var/lib/docker/volumes/nginx-vol/\_data",

"Destination": "/usr/share/nginx/html",

"Driver": "local",

"Mode": "",

"RW": false,

"Propagation": ""

}

],

Stop and remove the container, and remove the volume. Volume removal is a separate step.

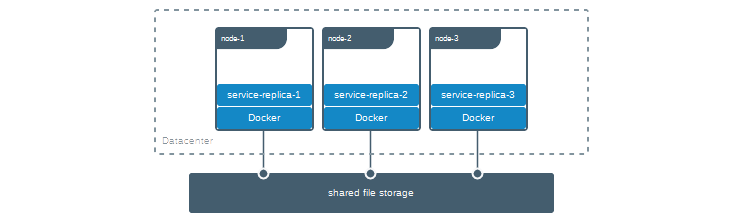
$ docker container stop nginxtest

$ docker container rm nginxtest

$ docker volume rm nginx-vol

## Share data among machines

When building fault-tolerant applications, you might need to configure multiple replicas of the same service to have access to the same files.



There are several ways to achieve this when developing your applications. One is to add logic to your application to store files on a cloud object storage system like Amazon S3. Another is to create volumes with a driver that supports writing files to an external storage system like NFS or Amazon S3.

Volume drivers allow you to abstract the underlying storage system from the application logic. For example, if your services use a volume with an NFS driver, you can update the services to use a different driver, as an example to store data in the cloud, without changing the application logic.

## Use a volume driver

When you create a volume using docker volume create, or when you start a container which uses a not-yet-created volume, you can specify a volume driver. The following examples use the vieux/sshfs volume driver, first when creating a standalone volume, and then when starting a container which creates a new volume.

### Initial set-up

This example assumes that you have two nodes, the first of which is a Docker host and can connect to the second using SSH.

On the Docker host, install the vieux/sshfs plugin:

$ docker plugin install --grant-all-permissions vieux/sshfs

### Create a volume using a volume driver

This example specifies a SSH password, but if the two hosts have shared keys configured, you can omit the password. Each volume driver may have zero or more configurable options, each of which is specified using an -o flag.

$ docker volume create --driver vieux/sshfs \

-o sshcmd=test@node2:/home/test \

-o password=testpassword \

sshvolume

### Start a container which creates a volume using a volume driver

This example specifies a SSH password, but if the two hosts have shared keys configured, you can omit the password. Each volume driver may have zero or more configurable options. **If the volume driver requires you to pass options, you must use the --mount flag to mount the volume, rather than -v.**

$ docker run -d \

--name sshfs-container \

--volume-driver vieux/sshfs \

--mount src=sshvolume,target=/app,volume-opt=sshcmd=test@node2:/home/test,volume-opt=password=testpassword \

nginx:latest

### Create a service which creates an NFS volume

This example shows how you can create an NFS volume when creating a service. This example uses 10.0.0.10 as the NFS server and /var/docker-nfs as the exported directory on the NFS server. Note that the volume driver specified is local.

#### NFSv3

$ docker service create -d \

--name nfs-service \

--mount 'type=volume,source=nfsvolume,target=/app,volume-driver=local,volume-opt=type=nfs,volume-opt=device=:/var/docker-nfs,volume-opt=o=addr=10.0.0.10' \

nginx:latest

#### NFSv4

$ docker service create -d \

--name nfs-service \

--mount 'type=volume,source=nfsvolume,target=/app,volume-driver=local,volume-opt=type=nfs,volume-opt=device=:/var/docker-nfs,"volume-opt=o=addr=10.0.0.10,rw,nfsvers=4,async"' \

nginx:latest

### Create CIFS/Samba volumes

You can mount a Samba share directly in docker without configuring a mount point on your host.

$ docker volume create \

--driver local \

--opt type=cifs \

--opt device=//uxxxxx.your-server.de/backup \

--opt o=addr=uxxxxx.your-server.de,username=uxxxxxxx,password=\*\*\*\*\*,file\_mode=0777,dir\_mode=0777 \

--name cif-volume

Notice the addr option is required if using a hostname instead of an IP so docker can perform the hostname lookup.

## Backup, restore, or migrate data volumes

Volumes are useful for backups, restores, and migrations. Use the --volumes-from flag to create a new container that mounts that volume.

### Back up a volume

For example, create a new container named dbstore:

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

Then in the next command, we:

* Launch a new container and mount the volume from the dbstore container
* Mount a local host directory as /backup
* Pass a command that tars the contents of the dbdata volume to a backup.tar file inside our /backup directory.

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

When the command completes and the container stops, we are left with a backup of our dbdata volume.

### Restore volume from backup

With the backup just created, you can restore it to the same container, or another that you made elsewhere.

For example, create a new container named dbstore2:

$ docker run -v /dbdata --name dbstore2 ubuntu /bin/bash

Then un-tar the backup file in the new container`s data volume:

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

You can use the techniques above to automate backup, migration and restore testing using your preferred tools.

## Remove volumes

A Docker data volume persists after a container is deleted. There are two types of volumes to consider:

* **Named volumes** have a specific source from outside the container, for example awesome:/bar.
* **Anonymous volumes** have no specific source so when the container is deleted, instruct the Docker Engine daemon to remove them.

### Remove anonymous volumes

To automatically remove anonymous volumes, use the --rm option. For example, this command creates an anonymous /foo volume. When the container is removed, the Docker Engine removes the /foo volume but not the awesome volume.

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

**Note:**

If another container binds the volumes with --volumes-from, the volume definitions are copied and the anonymous volume also stays after the first container is removed.

### Remove all volumes

To remove all unused volumes and free up space:

$ docker volume prune

# Set up Automated Builds

## How Automated Builds work

Docker Hub can automatically build images from source code in an external repository and automatically push the built image to your Docker repositories.

When you set up automated builds (also called autobuilds), you create a list of branches and tags that you want to build into Docker images. When you push code to a source code branch (for example in GitHub) for one of those listed image tags, the push uses a webhook to trigger a new build, which produces a Docker image. The built image is then pushed to the Docker Hub registry.

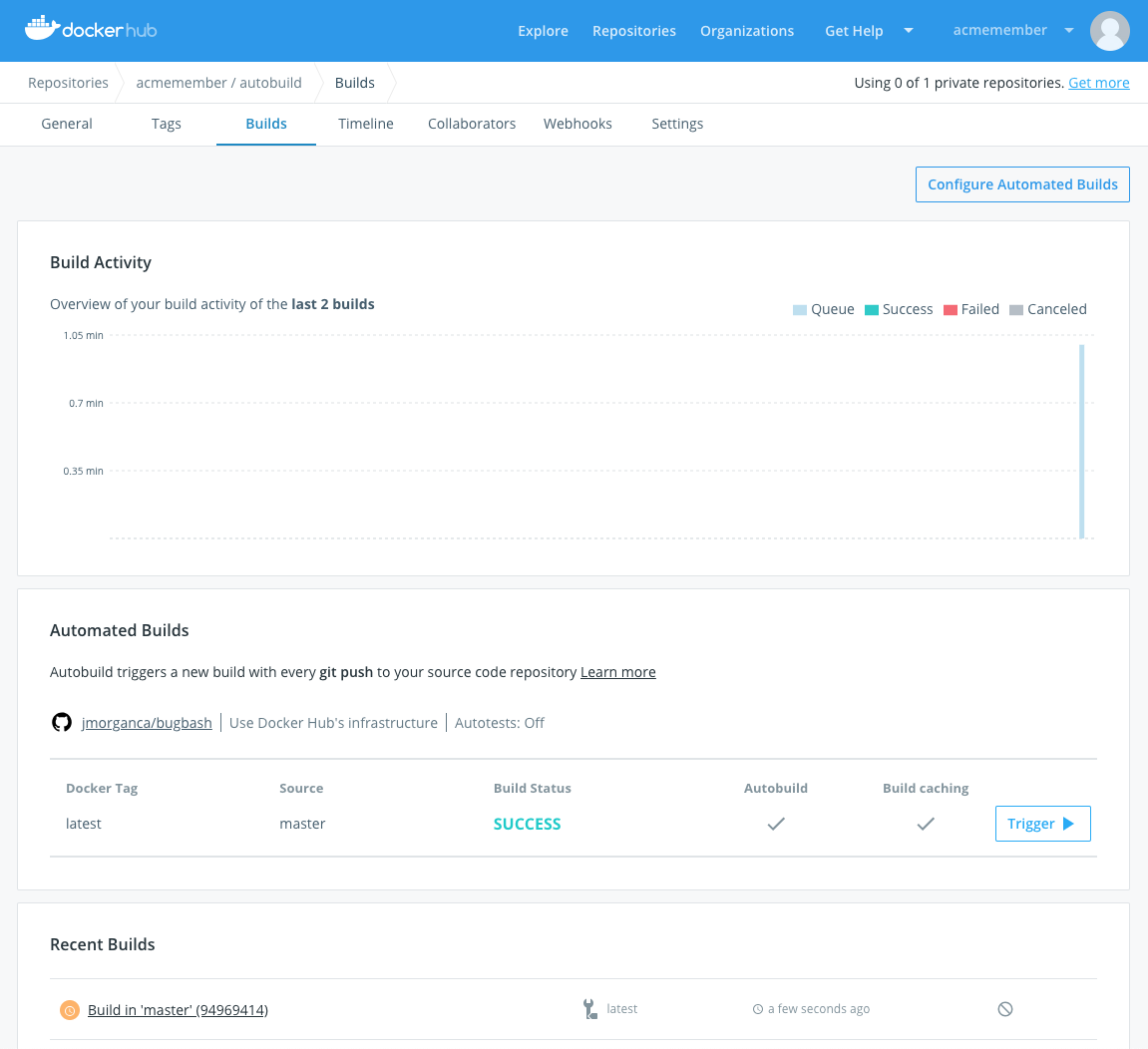
**Note**

You can still use docker push to push pre-built images to repositories with Automated Builds configured.

If you have automated tests configured, these run after building but before pushing to the registry. You can use these tests to create a continuous integration workflow where a build that fails its tests does not push the built image. Automated tests do not push images to the registry on their own. [Learn more about automated image testing here.](https://docs.docker.com/docker-hub/builds/automated-testing/)

Depending on your [plan](https://www.docker.com/pricing), you may get concurrent builds, which means that N autobuilds can be run at the same time. N is configured by the plan that you subscribe to. Once N+1 builds are running, any additional builds are queued to be run later.

The total number of pending builds in the queue is limited to 30 and further requests will be discarded. The number of concurrent builds for Pro is 5 and for Team and Business is 15.



## Configure automated build settings

You can configure repositories in Docker Hub so that they automatically build an image each time you push new code to your source provider. If you have [automated tests](https://docs.docker.com/docker-hub/builds/automated-testing/) configured, the new image is only pushed when the tests succeed.

Builds can be added to existing repositories, or added when you create a repository.

1. From the **Repositories** section, click into a repository to view its details.
2. Click the **Builds** tab.
3. If you are setting up automated builds for the first time, select the code repository service (GitHub or Bitbucket) where the image’s source code is stored. You may be redirected to the settings page to [link](https://docs.docker.com/docker-hub/builds/link-source/) the code repository service.

Otherwise, if you are editing the build settings for an existing automated build, click **Configure automated builds**.

1. Select the **source repository** to build the Docker images from.

You might need to specify an organization or user (the namespace) from the source code provider. Once you select a namespace, its source code repositories appear in the **Select repository** dropdown list.

1. Optionally, enable [autotests](https://docs.docker.com/docker-hub/builds/automated-testing/" \l "enable-automated-tests-on-a-repository).
2. Review the default **Build Rules**, and optionally click the **plus sign** to add and configure more build rules.

Build rules control what Docker Hub builds into images from the contents of the source code repository, and how the resulting images are tagged within the Docker repository.

A default build rule is set up for you, which you can edit or delete. This default set builds from the Branch in your source code repository called master, and creates a Docker image tagged with latest.

1. For each branch or tag, enable or disable the **Autobuild** toggle.

Only branches or tags with autobuild enabled are built, tested, and have the resulting image pushed to the repository. Branches with autobuild disabled are built for test purposes (if enabled at the repository level), but the built Docker image is not pushed to the repository.

1. For each branch or tag, enable or disable the **Build Caching** toggle.

[Build caching](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/" \l "leverage-build-cache) can save time if you are building a large image frequently or have many dependencies. You might want to leave build caching disabled to make sure all of your dependencies are resolved at build time, or if you have a large layer that is quicker to build locally.

1. Click **Save** to save the settings, or click **Save and build** to save and run an initial test.

A webhook is automatically added to your source code repository to notify Docker Hub on every push. Only pushes to branches that are listed as the source for one or more tags trigger a build.

### Set up build rules

By default when you set up automated builds, a basic build rule is created for you. This default rule watches for changes to the master branch in your source code repository, and builds the master branch into a Docker image tagged with latest.

In the **Build Rules** section, enter one or more sources to build.

For each source:

* Select the **Source type** to build either a **tag** or a **branch**. This tells the build system what to look for in the source code repository.
* Enter the name of the **Source** branch or tag you want to build.

The first time you configure automated builds, a default build rule is set up for you. This default set builds from the Branch in your source code called master, and creates a Docker image tagged with latest.

You can also use a regex to select which source branches or tags to build. To learn more, see [regexes](https://docs.docker.com/docker-hub/builds/" \l "regexes-and-automated-builds).

* Enter the tag to apply to Docker images built from this source.

If you configured a regex to select the source, you can reference the capture groups and use its result as part of the tag. To learn more, see [regexes](https://docs.docker.com/docker-hub/builds/" \l "regexes-and-automated-builds).

* Specify the **Dockerfile location** as a path relative to the root of the source code repository. If the Dockerfile is at the repository root, leave this path set to /.

**Note**

When Docker Hub pulls a branch from a source code repository, it performs a shallow clone (only the tip of the specified branch). Refer to [Advanced options for Autobuild and Autotest](https://docs.docker.com/docker-hub/builds/advanced/" \l "source-repository--branch-clones) for more information.

### Environment variables for builds

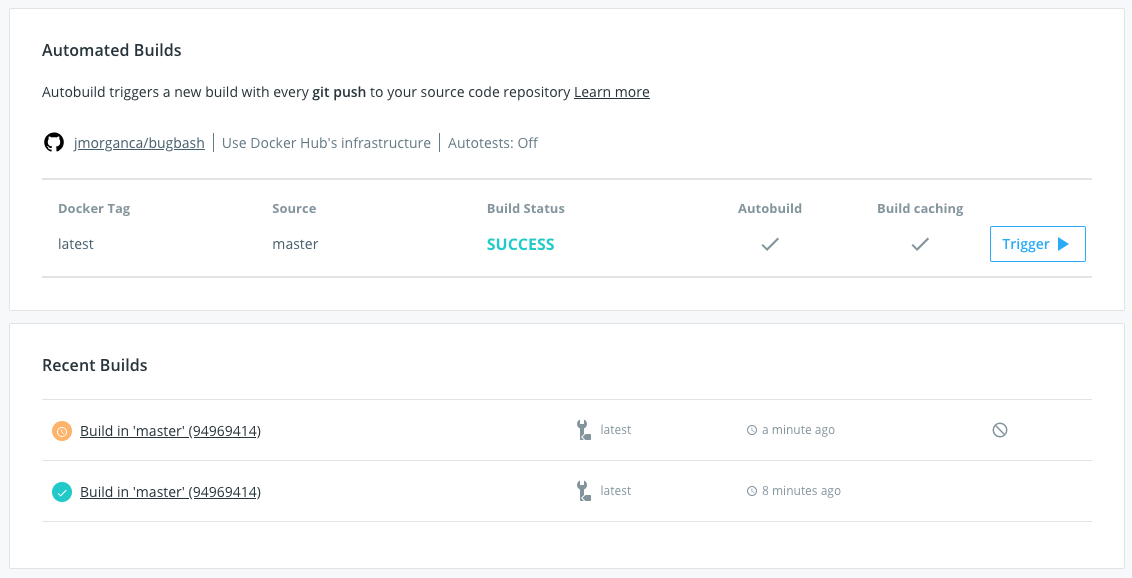
You can set the values for environment variables used in your build processes when you configure an automated build. Add your build environment variables by clicking the plus sign next to the **Build environment variables** section, and then entering a variable name and the value.

When you set variable values from the Docker Hub UI, they can be used by the commands you set in hooks files, but they are stored so that only users who have admin access to the Docker Hub repository can see their values. This means you can use them to safely store access tokens or other information that should remain secret.

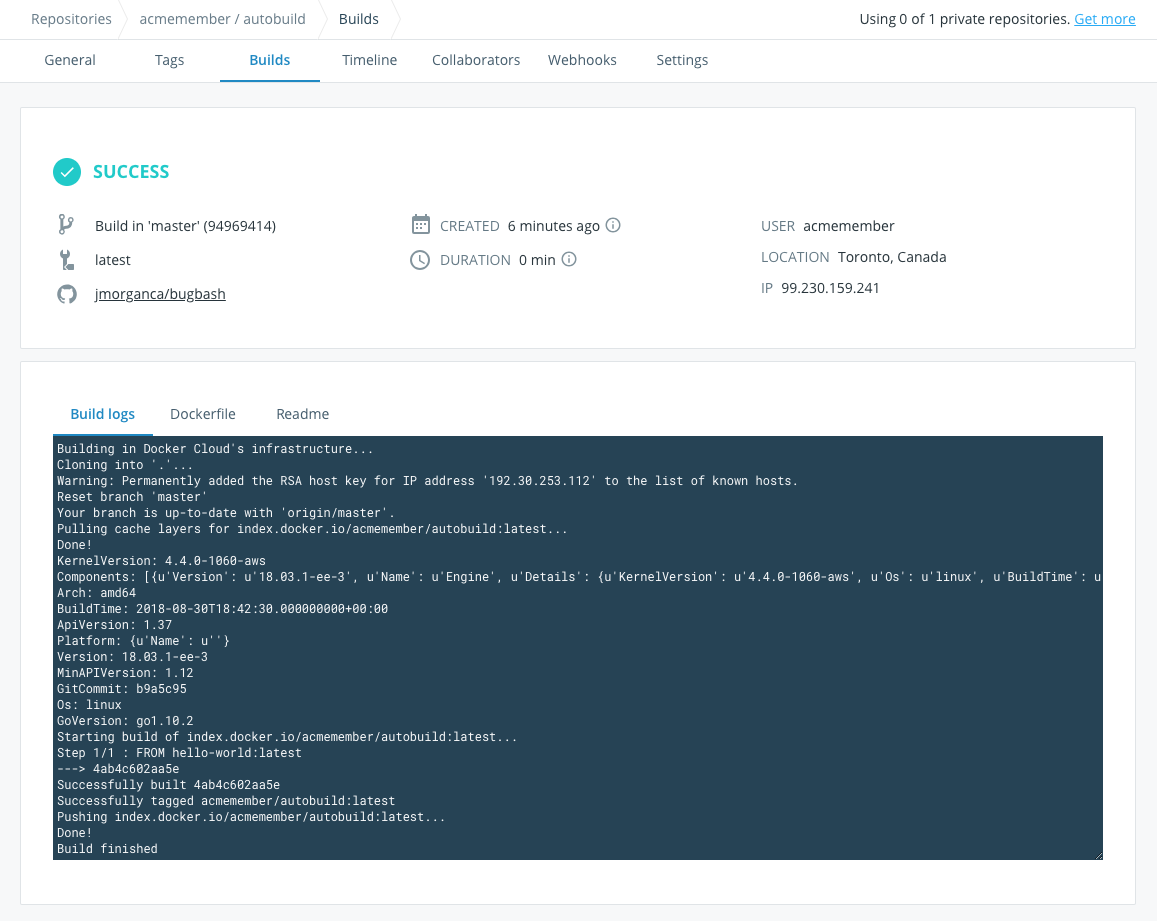
**Note**: The variables set on the build configuration screen are used during the build processes only and should not be confused with the environment values used by your service (for example to create service links).

## Check your active builds

A summary of a repository’s builds appears both on the repository **General** tab, and in the **Builds** tab. The **Builds** tab also displays a color coded bar chart of the build queue times and durations. Both views display the pending, in progress, successful, and failed builds for any tag of the repository.

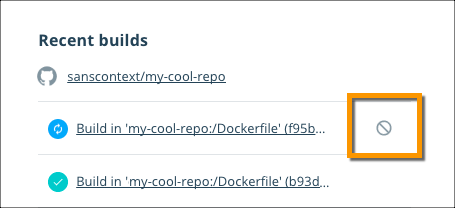


From either location, you can click a build job to view its build report. The build report shows information about the build job including the source repository and branch (or tag), the build duration, creation time and location, and the user namespace the build occurred in.

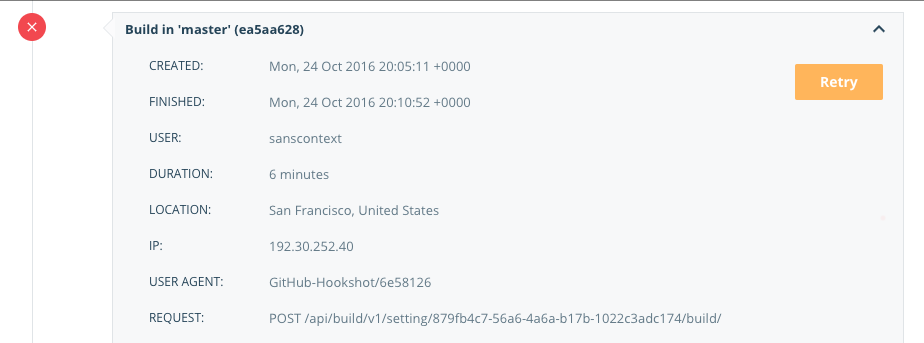


## Cancel or retry a build

While a build is queued or running, a **Cancel** icon appears next to its build report link on the General tab and on the Builds tab. You can also click the **Cancel** button from the build report page, or from the Timeline tab’s logs display for the build.



If a build fails, a **Retry** icon appears next to the build report line on the General and Builds tabs, and the build report page and Timeline logs also display a **Retry** button.



**Note**

If you are viewing the build details for a repository that belongs to an Organization, the Cancel and Retry buttons only appear if you have Read & Write access to the repository.

## Disable an automated build

Automated builds are enabled per branch or tag, and can be disabled and re-enabled easily. You might do this when you want to only build manually for a while, for example when you are doing major refactoring in your code. Disabling autobuilds does not disable [autotests](https://docs.docker.com/docker-hub/builds/automated-testing/).

To disable an automated build:

1. From the **Repositories** page, click into a repository, and click the **Builds** tab.
2. Click **Configure automated builds** to edit the repository’s build settings.
3. In the **Build Rules** section, locate the branch or tag you no longer want to automatically build.
4. Click the **autobuild** toggle next to the configuration line.

The toggle turns gray when disabled.

1. Click **Save** to save your changes.

## Advanced automated build options

At the minimum you need a build rule composed of a source branch (or tag) and destination Docker tag to set up an automated build. You can also change where the build looks for the Dockerfile, set a path to the files the build use (the build context), set up multiple static tags or branches to build from, and use regular expressions (regexes) to dynamically select source code to build and create dynamic tags.

All of these options are available from the **Build configuration** screen for each repository. Click **Repositories** from the left navigation, click the name of the repository you want to edit, click the **Builds** tab, and click **Configure Automated builds**.

### Tag and Branch builds

You can configure your automated builds so that pushes to specific branches or tags triggers a build.

1. In the **Build Rules** section, click the plus sign to add more sources to build.
2. Select the **Source type** to build: either a **tag** or a **branch**.

This tells the build system what type of source to look for in the code repository.

1. Enter the name of the **Source** branch or tag you want to build.

You can enter a name, or use a regex to match which source branch or tag names to build. To learn more, see [regexes](https://docs.docker.com/docker-hub/builds/" \l "regexes-and-automated-builds).

1. Enter the tag to apply to Docker images built from this source.

If you configured a regex to select the source, you can reference the capture groups and use its result as part of the tag. To learn more, see [regexes](https://docs.docker.com/docker-hub/builds/" \l "regexes-and-automated-builds).

1. Repeat steps 2 through 4 for each new build rule you set up.

### Set the build context and Dockerfile location

Depending on how the files are arranged in your source code repository, the files required to build your images may not be at the repository root. If that’s the case, you can specify a path where the build looks for the files.

The build context is the path to the files needed for the build, relative to the root of the repository. Enter the path to these files in the **Build context** field. Enter / to set the build context as the root of the source code repository.

**Note**

If you delete the default path / from the **Build context** field and leave it blank, the build system uses the path to the Dockerfile as the build context. However, to avoid confusion we recommend that you specify the complete path.

You can specify the **Dockerfile location** as a path relative to the build context. If the Dockerfile is at the root of the build context path, leave the Dockerfile path set to /. (If the build context field is blank, set the path to the Dockerfile from the root of the source repository.)

### Regexes and automated builds

You can specify a regular expression (regex) so that only matching branches or tags are built. You can also use the results of the regex to create the Docker tag that is applied to the built image.

You can use up to nine regular expression capture groups (expressions enclosed in parentheses) to select a source to build, and reference these in the **Docker Tag** field using {\1} through {\9}.

### Build images with BuildKit

Autobuilds use the BuildKit build system by default. If you want to use the legacy Docker build system, add the [environment variable](_blank) DOCKER\_BUILDKIT=0. Refer to the [build images with BuildKit](https://docs.docker.com/develop/develop-images/build_enhancements/) page for more information on BuildKit.

## Build repositories with linked private submodules

Docker Hub sets up a deploy key in your source code repository that allows it to clone the repository and build it; however this key only works for a single, specific code repository. If your source code repository uses private Git submodules (or requires that you clone other private repositories to build), Docker Hub cannot access these additional repos, your build cannot complete, and an error is logged in your build timeline.

To work around this, you can set up your automated build using the SSH\_PRIVATE environment variable to override the deployment key and grant Docker Hub’s build system access to the repositories.

**Note**

If you are using autobuild for teams, use [the process below](https://docs.docker.com/docker-hub/builds/" \l "service-users-for-team-autobuilds) instead, and configure a service user for your source code provider. You can also do this for an individual account to limit Docker Hub’s access to your source repositories.

1. Generate a SSH keypair that you use for builds only, and add the public key to your source code provider account.

This step is optional, but allows you to revoke the build-only keypair without removing other access.

1. Copy the private half of the keypair to your clipboard.
2. In Docker Hub, navigate to the build page for the repository that has linked private submodules. (If necessary, follow the steps [here](https://docs.docker.com/docker-hub/builds/" \l "configure-automated-build-settings) to configure the automated build.)
3. At the bottom of the screen, click the plus sign ( **+** ) next to **Build Environment variables**.
4. Enter SSH\_PRIVATE as the name for the new environment variable.
5. Paste the private half of the keypair into the **Value** field.
6. Click **Save**, or **Save and Build** to validate that the build now completes.

**Note**

You must configure your private git submodules using git clone over SSH (git@submodule.tld:some-submodule.git) rather than HTTPS.

## Autobuild for Teams

When you create an automated build repository in your own account namespace, you can start, cancel, and retry builds, and edit and delete your own repositories.

These same actions are also available for team repositories from Docker Hub if you are a member of the Organization’s Owners team. If you are a member of a team with write permissions you can start, cancel, and retry builds in your team’s repositories, but you cannot edit the team repository settings or delete the team repositories. If your user account has read permission, or if you’re a member of a team with read permission, you can view the build configuration including any testing settings.

| **Action/Permission** | **read** | **write** | **admin** | **owner** |
| --- | --- | --- | --- | --- |
| view build details | x | x | x | x |
| start, cancel, retry |  | x | x | x |
| edit build settings |  |  | x | x |
| delete build |  |  |  | x |

### Service users for team autobuilds

**Note**: Only members of the Owners team can set up automated builds for teams.

When you set up automated builds for teams, you grant Docker Hub access to your source code repositories using OAuth tied to a specific user account. This means that Docker Hub has access to everything that the linked source provider account can access.

For organizations and teams, we recommend creating a dedicated service account (or “machine user”) to grant access to the source provider. This ensures that no builds break as individual users’ access permissions change, and that an individual user’s personal projects are not exposed to an entire organization.

This service account should have access to any repositories to be built, and must have administrative access to the source code repositories so it can manage deploy keys. If needed, you can limit this account to only a specific set of repositories required for a specific build.

If you are building repositories with linked private submodules (private dependencies), you also need to add an override SSH\_PRIVATE environment variable to automated builds associated with the account.

1. Create a service user account on your source provider, and generate SSH keys for it.
2. Create a “build” team in your organization.
3. Ensure that the new “build” team has access to each repository and submodule you need to build.

Go to the repository’s **Settings** page. On GitHub, add the new “build” team to the list of **Collaborators and Teams**. On Bitbucket, add the “build” team to the list of approved users on the **Access management** screen.

1. Add the service user to the “build” team on the source provider.
2. Log in to Docker Hub as a member of the Owners team, switch to the organization, and follow the instructions to [link to source code repository](https://docs.docker.com/docker-hub/builds/link-source/) using the service account.

**Note**: You may need to log out of your individual account on the source code provider to create the link to the service account.

1. Optionally, use the SSH keys you generated to set up any builds with private submodules, using the service account and [the instructions above](https://docs.docker.com/docker-hub/builds/" \l "build-repositories-with-linked-private-submodules).

## What’s Next?

### Customize your build process

Additional advanced options are available for customizing your automated builds, including utility environment variables, hooks, and build phase overrides. To learn more see [Advanced options for Autobuild and Autotest](https://docs.docker.com/docker-hub/builds/advanced/).

### Add automated tests

To test your code before the image is pushed, you can use Docker Hub’s [Autotest](https://docs.docker.com/docker-hub/builds/automated-testing/) feature which integrates seamlessly with autobuild and autoredeploy.

**Note**

While the Autotest feature builds an image for testing purposes, it does not push the resulting image to Docker Hub.

| Command | Description |
| --- | --- |
| [docker attach](https://docs.docker.com/engine/reference/commandline/attach/) | Attach local standard input, output, and error streams to a running container |
| [docker build](https://docs.docker.com/engine/reference/commandline/build/) | Build an image from a Dockerfile |
| [docker builder](https://docs.docker.com/engine/reference/commandline/builder/) | Manage builds |
| [docker checkpoint](https://docs.docker.com/engine/reference/commandline/checkpoint/) | Manage checkpoints |
| [docker commit](https://docs.docker.com/engine/reference/commandline/commit/) | Create a new image from a container’s changes |
| [docker config](https://docs.docker.com/engine/reference/commandline/config/) | Manage Docker configs |
| [docker container](https://docs.docker.com/engine/reference/commandline/container/) | Manage containers |
| [docker context](https://docs.docker.com/engine/reference/commandline/context/) | Manage contexts |
| [docker cp](https://docs.docker.com/engine/reference/commandline/cp/) | Copy files/folders between a container and the local filesystem |
| [docker create](https://docs.docker.com/engine/reference/commandline/create/) | Create a new container |
| [docker diff](https://docs.docker.com/engine/reference/commandline/diff/) | Inspect changes to files or directories on a container’s filesystem |
| [docker events](https://docs.docker.com/engine/reference/commandline/events/) | Get real time events from the server |
| [docker exec](https://docs.docker.com/engine/reference/commandline/exec/) | Run a command in a running container |
| [docker export](https://docs.docker.com/engine/reference/commandline/export/) | Export a container’s filesystem as a tar archive |
| [docker history](https://docs.docker.com/engine/reference/commandline/history/) | Show the history of an image |
| [docker image](https://docs.docker.com/engine/reference/commandline/image/) | Manage images |
| [docker images](https://docs.docker.com/engine/reference/commandline/images/) | List images |
| [docker import](https://docs.docker.com/engine/reference/commandline/import/) | Import the contents from a tarball to create a filesystem image |
| [docker info](https://docs.docker.com/engine/reference/commandline/info/) | Display system-wide information |
| [docker inspect](https://docs.docker.com/engine/reference/commandline/inspect/) | Return low-level information on Docker objects |
| [docker kill](https://docs.docker.com/engine/reference/commandline/kill/) | Kill one or more running containers |
| [docker load](https://docs.docker.com/engine/reference/commandline/load/) | Load an image from a tar archive or STDIN |
| [docker login](https://docs.docker.com/engine/reference/commandline/login/) | Log in to a Docker registry |
| [docker logout](https://docs.docker.com/engine/reference/commandline/logout/) | Log out from a Docker registry |
| [docker logs](https://docs.docker.com/engine/reference/commandline/logs/) | Fetch the logs of a container |
| [docker manifest](https://docs.docker.com/engine/reference/commandline/manifest/) | Manage Docker image manifests and manifest lists |
| [docker network](https://docs.docker.com/engine/reference/commandline/network/) | Manage networks |
| [docker node](https://docs.docker.com/engine/reference/commandline/node/) | Manage Swarm nodes |
| [docker pause](https://docs.docker.com/engine/reference/commandline/pause/) | Pause all processes within one or more containers |
| [docker plugin](https://docs.docker.com/engine/reference/commandline/plugin/) | Manage plugins |
| [docker port](https://docs.docker.com/engine/reference/commandline/port/) | List port mappings or a specific mapping for the container |
| [docker ps](https://docs.docker.com/engine/reference/commandline/ps/) | List containers |
| [docker pull](https://docs.docker.com/engine/reference/commandline/pull/) | Pull an image or a repository from a registry |
| [docker push](https://docs.docker.com/engine/reference/commandline/push/) | Push an image or a repository to a registry |
| [docker rename](https://docs.docker.com/engine/reference/commandline/rename/) | Rename a container |
| [docker restart](https://docs.docker.com/engine/reference/commandline/restart/) | Restart one or more containers |
| [docker rm](https://docs.docker.com/engine/reference/commandline/rm/) | Remove one or more containers |
| [docker rmi](https://docs.docker.com/engine/reference/commandline/rmi/) | Remove one or more images |
| [docker run](https://docs.docker.com/engine/reference/commandline/run/) | Run a command in a new container |
| [docker save](https://docs.docker.com/engine/reference/commandline/save/) | Save one or more images to a tar archive (streamed to STDOUT by default) |
| [docker search](https://docs.docker.com/engine/reference/commandline/search/) | Search the Docker Hub for images |
| [docker secret](https://docs.docker.com/engine/reference/commandline/secret/) | Manage Docker secrets |
| [docker service](https://docs.docker.com/engine/reference/commandline/service/) | Manage services |
| [docker stack](https://docs.docker.com/engine/reference/commandline/stack/) | Manage Docker stacks |
| [docker start](https://docs.docker.com/engine/reference/commandline/start/) | Start one or more stopped containers |
| [docker stats](https://docs.docker.com/engine/reference/commandline/stats/) | Display a live stream of container(s) resource usage statistics |
| [docker stop](https://docs.docker.com/engine/reference/commandline/stop/) | Stop one or more running containers |
| [docker swarm](https://docs.docker.com/engine/reference/commandline/swarm/) | Manage Swarm |
| [docker system](https://docs.docker.com/engine/reference/commandline/system/) | Manage Docker |
| [docker tag](https://docs.docker.com/engine/reference/commandline/tag/) | Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE |
| [docker top](https://docs.docker.com/engine/reference/commandline/top/) | Display the running processes of a container |
| [docker trust](https://docs.docker.com/engine/reference/commandline/trust/) | Manage trust on Docker images |
| [docker unpause](https://docs.docker.com/engine/reference/commandline/unpause/) | Unpause all processes within one or more containers |
| [docker update](https://docs.docker.com/engine/reference/commandline/update/) | Update configuration of one or more containers |
| [docker version](https://docs.docker.com/engine/reference/commandline/version/) | Show the Docker version information |
| [docker volume](https://docs.docker.com/engine/reference/commandline/volume/) | Manage volumes |
| [docker wait](https://docs.docker.com/engine/reference/commandline/wait/) | Block until one or more containers stop, then print their exit codes |

# Overview of Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration

Compose, see [the list of features](https://docs.docker.com/compose/" \l "features).

Compose works in all environments: production, staging, development, testing, as well as CI workflows. You can learn more about each case in [Common Use Cases](https://docs.docker.com/compose/" \l "common-use-cases).

Using Compose is basically a three-step process:

1. Define your app’s environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Run docker compose up and the [Docker compose command](https://docs.docker.com/compose/" \l "compose-v2-and-the-new-docker-compose-command) starts and runs your entire app. You can alternatively run docker-compose up using the docker-compose binary.

A docker-compose.yml looks like this:

version: "3.9" # optional since v1.27.0

services:

web:

build: .

ports:

- "8000:5000"

volumes:

- .:/code

- logvolume01:/var/log

links:

- redis

redis:

image: redis

volumes:

logvolume01: {}

For more information about the Compose file, see the [Compose file reference](https://docs.docker.com/compose/compose-file/).

Compose has commands for managing the whole lifecycle of your application:

* Start, stop, and rebuild services
* View the status of running services
* Stream the log output of running services
* Run a one-off command on a service

## Compose V2 and the new docker compose command

**Important**

The new Compose V2, which supports the compose command as part of the Docker CLI, is now available.

Compose V2 integrates compose functions into the Docker platform, continuing to support most of the previous docker-compose features and flags. You can run Compose V2 by replacing the hyphen (-) with a space, using docker compose, instead of docker-compose.

If you rely on using Docker Compose as docker-compose (with a hyphen), you can set up Compose V2 to act as a drop-in replacement of the previous docker-compose. Refer to the [Installing Compose](https://docs.docker.com/compose/install/) section for detailed instructions.

## Context of Docker Compose evolution

Introduction of the [Compose specification](https://github.com/compose-spec/compose-spec) makes a clean distinction between the Compose YAML file model and the docker-compose implementation. Making this change has enabled a number of enhancements, including adding the compose command directly into the Docker CLI, being able to “up” a Compose application on cloud platforms by simply switching the Docker context, and launching of [Amazon ECS](https://docs.docker.com/cloud/ecs-integration/) and [Microsoft ACI](https://docs.docker.com/cloud/aci-integration/). As the Compose specification evolves, new features land faster in the Docker CLI.

Compose V2 relies directly on the compose-go bindings which are maintained as part of the specification. This allows us to include community proposals, experimental implementations by the Docker CLI and/or Engine, and deliver features faster to users. Compose V2 also supports some of the newer additions to the specification, such as [profiles](https://docs.docker.com/compose/profiles/) and [GPU](https://docs.docker.com/compose/gpu-support/) devices.

Compose V2 has been re-written in [Go](https://go.dev/), which improves integration with other Docker command-line features, and allows it to run natively on [macOS on Apple silicon](https://docs.docker.com/desktop/mac/apple-silicon/), Windows, and Linux, without dependencies such as Python.

For more information about compatibility with the compose v1 command-line, see the [docker-compose compatibility list](https://docs.docker.com/compose/cli-command-compatibility/).

## Features

The features of Compose that make it effective are:

* [Multiple isolated environments on a single host](https://docs.docker.com/compose/" \l "multiple-isolated-environments-on-a-single-host)
* [Preserve volume data when containers are created](https://docs.docker.com/compose/" \l "preserve-volume-data-when-containers-are-created)
* [Only recreate containers that have changed](https://docs.docker.com/compose/" \l "only-recreate-containers-that-have-changed)
* [Variables and moving a composition between environments](https://docs.docker.com/compose/" \l "variables-and-moving-a-composition-between-environments)

### Multiple isolated environments on a single host

Compose uses a project name to isolate environments from each other. You can make use of this project name in several different contexts:

* on a dev host, to create multiple copies of a single environment, such as when you want to run a stable copy for each feature branch of a project
* on a CI server, to keep builds from interfering with each other, you can set the project name to a unique build number
* on a shared host or dev host, to prevent different projects, which may use the same service names, from interfering with each other

The default project name is the basename of the project directory. You can set a custom project name by using the [-p command line option](https://docs.docker.com/compose/reference/) or the [COMPOSE\_PROJECT\_NAME](https://docs.docker.com/compose/reference/envvars/" \l "compose_project_name) environment variable.

The default project directory is the base directory of the Compose file. A custom value for it can be defined with the --project-directory command line option.

### Preserve volume data when containers are created

Compose preserves all volumes used by your services. When docker-compose up runs, if it finds any containers from previous runs, it copies the volumes from the old container to the new container. This process ensures that any data you’ve created in volumes isn’t lost.

If you use docker-compose on a Windows machine, see [Environment variables](https://docs.docker.com/compose/reference/envvars/) and adjust the necessary environment variables for your specific needs.

### Only recreate containers that have changed

Compose caches the configuration used to create a container. When you restart a service that has not changed, Compose re-uses the existing containers. Re-using containers means that you can make changes to your environment very quickly.

### Variables and moving a composition between environments

Compose supports variables in the Compose file. You can use these variables to customize your composition for different environments, or different users. See [Variable substitution](https://docs.docker.com/compose/compose-file/compose-file-v3/" \l "variable-substitution) for more details.

You can extend a Compose file using the extends field or by creating multiple Compose files. See [extends](https://docs.docker.com/compose/extends/) for more details.

## Common use cases

Compose can be used in many different ways. Some common use cases are outlined below.

### Development environments

When you’re developing software, the ability to run an application in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.

The [Compose file](https://docs.docker.com/compose/compose-file/) provides a way to document and configure all of the application’s service dependencies (databases, queues, caches, web service APIs, etc). Using the Compose command line tool you can create and start one or more containers for each dependency with a single command (docker-compose up).

Together, these features provide a convenient way for developers to get started on a project. Compose can reduce a multi-page “developer getting started guide” to a single machine readable Compose file and a few commands.

### Automated testing environments

An important part of any Continuous Deployment or Continuous Integration process is the automated test suite. Automated end-to-end testing requires an environment in which to run tests. Compose provides a convenient way to create and destroy isolated testing environments for your test suite. By defining the full environment in a [Compose file](https://docs.docker.com/compose/compose-file/), you can create and destroy these environments in just a few commands:

$ docker-compose up -d

$ ./run\_tests

$ docker-compose down

### Single host deployments

Compose has traditionally been focused on development and testing workflows, but with each release we’re making progress on more production-oriented features.

For details on using production-oriented features, see [compose in production](https://docs.docker.com/compose/production/) in this documentation.

# docker swarm

Swarm is a container orchestration tool that allows a user to manager multiple containers across multiple host machines.

Creates product services at scale.

Manage Swarm

**Swarm** This command works with the Swarm orchestrator.

## Usage

$ docker swarm COMMAND

## Description

Manage the swarm.

## Child commands

| Command | Description |
| --- | --- |
| [docker swarm ca](https://docs.docker.com/engine/reference/commandline/swarm_ca/) | Display and rotate the root CA |
| [docker swarm init](https://docs.docker.com/engine/reference/commandline/swarm_init/) | Initialize a swarm |
| [docker swarm join](https://docs.docker.com/engine/reference/commandline/swarm_join/) | Join a swarm as a node and/or manager |
| [docker swarm join-token](https://docs.docker.com/engine/reference/commandline/swarm_join-token/) | Manage join tokens |
| [docker swarm leave](https://docs.docker.com/engine/reference/commandline/swarm_leave/) | Leave the swarm |
| [docker swarm unlock](https://docs.docker.com/engine/reference/commandline/swarm_unlock/) | Unlock swarm |
| [docker swarm unlock-key](https://docs.docker.com/engine/reference/commandline/swarm_unlock-key/) | Manage the unlock key |
| [docker swarm update](https://docs.docker.com/engine/reference/commandline/swarm_update/) | Update the swarm |

# Getting started with swarm mode

Estimated reading time: 3 minutes

This tutorial introduces you to the features of Docker Engine Swarm mode. You may want to familiarize yourself with the [key concepts](https://docs.docker.com/engine/swarm/key-concepts/) before you begin.

The tutorial guides you through the following activities:

* initializing a cluster of Docker Engines in swarm mode
* adding nodes to the swarm
* deploying application services to the swarm
* managing the swarm once you have everything running

This tutorial uses Docker Engine CLI commands entered on the command line of a terminal window.

If you are brand new to Docker, see [About Docker Engine](https://docs.docker.com/engine/).

## Set up

To run this tutorial, you need the following:

* [three Linux hosts which can communicate over a network, with Docker installed](https://docs.docker.com/engine/swarm/swarm-tutorial/" \l "three-networked-host-machines)
* [the IP address of the manager machine](https://docs.docker.com/engine/swarm/swarm-tutorial/" \l "the-ip-address-of-the-manager-machine)
* [open ports between the hosts](https://docs.docker.com/engine/swarm/swarm-tutorial/" \l "open-protocols-and-ports-between-the-hosts)

### Three networked host machines

This tutorial requires three Linux hosts which have Docker installed and can communicate over a network. These can be physical machines, virtual machines, Amazon EC2 instances, or hosted in some other way. Check out [Getting started - Swarms](https://docs.docker.com/get-started/swarm-deploy/" \l "prerequisites) for one possible set-up for the hosts.

One of these machines is a manager (called manager1) and two of them are workers (worker1 and worker2).

**Note**: You can follow many of the tutorial steps to test single-node swarm as well, in which case you need only one host. Multi-node commands do not work, but you can initialize a swarm, create services, and scale them.

#### Install Docker Engine on Linux machines

If you are using Linux based physical computers or cloud-provided computers as hosts, simply follow the [Linux install instructions](https://docs.docker.com/engine/install/) for your platform. Spin up the three machines, and you are ready. You can test both single-node and multi-node swarm scenarios on Linux machines.

#### Use Docker Desktop for Mac or Docker Desktop for Windows

Alternatively, install the latest [Docker Desktop for Mac](https://docs.docker.com/desktop/mac/) or [Docker Desktop for Windows](https://docs.docker.com/desktop/windows/) application on one computer. You can test both single-node and multi-node swarm from this computer.

* You can use Docker Desktop for Mac or Windows to test single-node features of swarm mode, including initializing a swarm with a single node, creating services, and scaling services.
* Currently, you cannot use Docker Desktop for Mac or Docker Desktop for Windows alone to test a multi-node swarm, but many examples are applicable to a single-node Swarm setup.

### The IP address of the manager machine

The IP address must be assigned to a network interface available to the host operating system. All nodes in the swarm need to connect to the manager at the IP address.

Because other nodes contact the manager node on its IP address, you should use a fixed IP address.

You can run ifconfig on Linux or macOS to see a list of the available network interfaces.

The tutorial uses manager1 : 192.168.99.100.

### Open protocols and ports between the hosts

The following ports must be available. On some systems, these ports are open by default.

* **TCP port 2377** for cluster management communications
* **TCP** and **UDP port 7946** for communication among nodes
* **UDP port 4789** for overlay network traffic

If you plan on creating an overlay network with encryption (--opt encrypted), you also need to ensure **ip protocol 50** (**ESP**) traffic is allowed.

## What’s next?

After you have set up your environment, you are ready to [create a swarm](https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/).

[tutorial](https://docs.docker.com/search/?q=tutorial), [cluster management](https://docs.docker.com/search/?q=cluster management), [swarm mode](https://docs.docker.com/search/?q=swarm mode)

# Create a swarm

Estimated reading time: 2 minutes

After you complete the [tutorial setup](https://docs.docker.com/engine/swarm/swarm-tutorial/) steps, you’re ready to create a swarm. Make sure the Docker Engine daemon is started on the host machines.

1. Open a terminal and ssh into the machine where you want to run your manager node. This tutorial uses a machine named manager1. If you use Docker Machine, you can connect to it via SSH using the following command:

$ docker-machine ssh manager1

1. Run the following command to create a new swarm:

$ docker swarm init --advertise-addr <MANAGER-IP>

**Note**: If you are using Docker Desktop for Mac or Docker Desktop for Windows to test single-node swarm, simply run docker swarm init with no arguments. There is no need to specify --advertise-addr in this case. To learn more, see the topic on how to [Use Docker Desktop for Mac or Docker Desktop for Windows](https://docs.docker.com/engine/swarm/swarm-tutorial/" \l "use-docker-desktop-for-mac-or-docker-desktop-for-windows) with Swarm.

In the tutorial, the following command creates a swarm on the manager1 machine:

$ docker swarm init --advertise-addr 192.168.99.100

Swarm initialized: current node (dxn1zf6l61qsb1josjja83ngz) is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join \

--token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \

192.168.99.100:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

The --advertise-addr flag configures the manager node to publish its address as 192.168.99.100. The other nodes in the swarm must be able to access the manager at the IP address.

The output includes the commands to join new nodes to the swarm. Nodes will join as managers or workers depending on the value for the --token flag.

1. Run docker info to view the current state of the swarm:

$ docker info

Containers: 2

Running: 0

Paused: 0

Stopped: 2

...snip...

Swarm: active

NodeID: dxn1zf6l61qsb1josjja83ngz

Is Manager: true

Managers: 1

Nodes: 1

...snip...

1. Run the docker node ls command to view information about nodes:

$ docker node ls

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS

dxn1zf6l61qsb1josjja83ngz \* manager1 Ready Active Leader

The \* next to the node ID indicates that you’re currently connected on this node.

Docker Engine swarm mode automatically names the node for the machine host name. The tutorial covers other columns in later steps.

## What’s next?

In the next section of the tutorial, we [add two more nodes](https://docs.docker.com/engine/swarm/swarm-tutorial/add-nodes/) to the cluster.

[tutorial](https://docs.docker.com/search/?q=tutorial), [cluster management](https://docs.docker.com/search/?q=cluster management), [swarm mode](https://docs.docker.com/search/?q=swarm mode)

# Add nodes to the swarm

Estimated reading time: 2 minutes

Once you’ve [created a swarm](https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/) with a manager node, you’re ready to add worker nodes.

1. Open a terminal and ssh into the machine where you want to run a worker node. This tutorial uses the name worker1.
2. Run the command produced by the docker swarm init output from the [Create a swarm](https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/) tutorial step to create a worker node joined to the existing swarm:

$ docker swarm join \

--token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \

192.168.99.100:2377

This node joined a swarm as a worker.

If you don’t have the command available, you can run the following command on a manager node to retrieve the join command for a worker:

$ docker swarm join-token worker

To add a worker to this swarm, run the following command:

docker swarm join \

--token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \

192.168.99.100:2377

1. Open a terminal and ssh into the machine where you want to run a second worker node. This tutorial uses the name worker2.
2. Run the command produced by the docker swarm init output from the [Create a swarm](https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/) tutorial step to create a second worker node joined to the existing swarm:

$ docker swarm join \

--token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \

192.168.99.100:2377

This node joined a swarm as a worker.

1. Open a terminal and ssh into the machine where the manager node runs and run the docker node ls command to see the worker nodes:

$ docker node ls

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS

03g1y59jwfg7cf99w4lt0f662 worker2 Ready Active

9j68exjopxe7wfl6yuxml7a7j worker1 Ready Active

dxn1zf6l61qsb1josjja83ngz \* manager1 Ready Active Leader

The MANAGER column identifies the manager nodes in the swarm. The empty status in this column for worker1 and worker2 identifies them as worker nodes.

Swarm management commands like docker node ls only work on manager nodes.

## What’s next?

Now your swarm consists of a manager and two worker nodes. In the next step of the tutorial, you [deploy a service](https://docs.docker.com/engine/swarm/swarm-tutorial/deploy-service/) to the swarm.

[tutorial](https://docs.docker.com/search/?q=tutorial), [cluster management](https://docs.docker.com/search/?q=cluster management), [swarm](https://docs.docker.com/search/?q=swarm)

# Deploy a service to the swarm

After you [create a swarm](https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/), you can deploy a service to the swarm. For this tutorial, you also [added worker nodes](https://docs.docker.com/engine/swarm/swarm-tutorial/add-nodes/), but that is not a requirement to deploy a service.

1. Open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named manager1.
2. Run the following command:

$ docker service create --replicas 1 --name helloworld alpine ping docker.com

9uk4639qpg7npwf3fn2aasksr

* + The docker service create command creates the service.
  + The --name flag names the service helloworld.
  + The --replicas flag specifies the desired state of 1 running instance.
  + The arguments alpine ping docker.com define the service as an Alpine Linux container that executes the command ping docker.com.

1. Run docker service ls to see the list of running services:

$ docker service ls

ID NAME SCALE IMAGE COMMAND

9uk4639qpg7n helloworld 1/1 alpine ping docker.com

## What’s next?

Now you’ve deployed a service to the swarm, you’re ready to [inspect the service](https://docs.docker.com/engine/swarm/swarm-tutorial/inspect-service/).

[tutorial](https://docs.docker.com/search/?q=tutorial), [cluster management](https://docs.docker.com/search/?q=cluster management), [swarm mode](https://docs.docker.com/search/?q=swarm mode)

# Inspect a service on the swarm

Estimated reading time: 2 minutes

When you have [deployed a service](https://docs.docker.com/engine/swarm/swarm-tutorial/deploy-service/) to your swarm, you can use the Docker CLI to see details about the service running in the swarm.

1. If you haven’t already, open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named manager1.
2. Run docker service inspect --pretty <SERVICE-ID> to display the details about a service in an easily readable format.

To see the details on the helloworld service:

[manager1]$ docker service inspect --pretty helloworld

ID: 9uk4639qpg7npwf3fn2aasksr

Name: helloworld

Service Mode: REPLICATED

Replicas: 1

Placement:

UpdateConfig:

Parallelism: 1

ContainerSpec:

Image: alpine

Args: ping docker.com

Resources:

Endpoint Mode: vip

**Tip**: To return the service details in json format, run the same command without the --pretty flag.

[manager1]$ docker service inspect helloworld

[

{

"ID": "9uk4639qpg7npwf3fn2aasksr",

"Version": {

"Index": 418

},

"CreatedAt": "2016-06-16T21:57:11.622222327Z",

"UpdatedAt": "2016-06-16T21:57:11.622222327Z",

"Spec": {

"Name": "helloworld",

"TaskTemplate": {

"ContainerSpec": {

"Image": "alpine",

"Args": [

"ping",

"docker.com"

]

},

"Resources": {

"Limits": {},

"Reservations": {}

},

"RestartPolicy": {

"Condition": "any",

"MaxAttempts": 0

},

"Placement": {}

},

"Mode": {

"Replicated": {

"Replicas": 1

}

},

"UpdateConfig": {

"Parallelism": 1

},

"EndpointSpec": {

"Mode": "vip"

}

},

"Endpoint": {

"Spec": {}

}

}

]

1. Run docker service ps <SERVICE-ID> to see which nodes are running the service:

[manager1]$ docker service ps helloworld

NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

helloworld.1.8p1vev3fq5zm0mi8g0as41w35 alpine worker2 Running Running 3 minutes

In this case, the one instance of the helloworld service is running on the worker2 node. You may see the service running on your manager node. By default, manager nodes in a swarm can execute tasks just like worker nodes.

Swarm also shows you the DESIRED STATE and CURRENT STATE of the service task so you can see if tasks are running according to the service definition.

1. Run docker ps on the node where the task is running to see details about the container for the task.

**Tip**: If helloworld is running on a node other than your manager node, you must ssh to that node.

[worker2]$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

# Scale the service in the swarm

Once you have [deployed a service](https://docs.docker.com/engine/swarm/swarm-tutorial/deploy-service/) to a swarm, you are ready to use the Docker CLI to scale the number of containers in the service. Containers running in a service are called “tasks.”

1. If you haven’t already, open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named manager1.
2. Run the following command to change the desired state of the service running in the swarm:

$ docker service scale <SERVICE-ID>=<NUMBER-OF-TASKS>

For example:

$ docker service scale helloworld=5

helloworld scaled to 5

1. Run docker service ps <SERVICE-ID> to see the updated task list:

$ docker service ps helloworld

NAME IMAGE NODE DESIRED STATE CURRENT STATE

helloworld.1.8p1vev3fq5zm0mi8g0as41w35 alpine worker2 Running Running 7 minutes

helloworld.2.c7a7tcdq5s0uk3qr88mf8xco6 alpine worker1 Running Running 24 seconds

helloworld.3.6crl09vdcalvtfehfh69ogfb1 alpine worker1 Running Running 24 seconds

helloworld.4.auky6trawmdlcne8ad8phb0f1 alpine manager1 Running Running 24 seconds

helloworld.5.ba19kca06l18zujfwxyc5lkyn alpine worker2 Running Running 24 seconds

You can see that swarm has created 4 new tasks to scale to a total of 5 running instances of Alpine Linux. The tasks are distributed between the three nodes of the swarm. One is running on manager1.

1. Run docker ps to see the containers running on the node where you’re connected. The following example shows the tasks running on manager1:

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

528d68040f95 alpine:latest "ping docker.com" About a minute ago Up About a minute helloworld.4.auky6trawmdlcne8ad8phb0f1

If you want to see the containers running on other nodes, ssh into those nodes and run the docker ps command.

## What’s next?

At this point in the tutorial, you’re finished with the helloworld service. The next step shows how to [delete the service](https://docs.docker.com/engine/swarm/swarm-tutorial/delete-service/).

[tutorial](https://docs.docker.com/search/?q=tutorial), [cluster management](https://docs.docker.com/search/?q=cluster management), [swarm mode](https://docs.docker.com/search/?q=swarm mode), [scale](https://docs.docker.com/search/?q=scale)

# Delete the service running on the swarm

The remaining steps in the tutorial don’t use the helloworld service, so now you can delete the service from the swarm.

1. If you haven’t already, open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named manager1.
2. Run docker service rm helloworld to remove the helloworld service.

$ docker service rm helloworld

helloworld

1. Run docker service inspect <SERVICE-ID> to verify that the swarm manager removed the service. The CLI returns a message that the service is not found:

$ docker service inspect helloworld

[]

Error: no such service: helloworld

1. Even though the service no longer exists, the task containers take a few seconds to clean up. You can use docker ps on the nodes to verify when the tasks have been removed.

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

db1651f50347 alpine:latest "ping docker.com" 44 minutes ago Up 46 seconds helloworld.5.9lkmos2beppihw95vdwxy1j3w

43bf6e532a92 alpine:latest "ping docker.com" 44 minutes ago Up 46 seconds helloworld.3.a71i8rp6fua79ad43ycocl4t2

5a0fb65d8fa7 alpine:latest "ping docker.com" 44 minutes ago Up 45 seconds helloworld.2.2jpgensh7d935qdc857pxulfr

afb0ba67076f alpine:latest "ping docker.com" 44 minutes ago Up 46 seconds helloworld.4.1c47o7tluz7drve4vkm2m5olx

688172d3bfaa alpine:latest "ping docker.com" 45 minutes ago Up About a minute helloworld.1.74nbhb3fhud8jfrhigd7s29we

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

## What’s next?

In the next step of the tutorial, you set up a new service and apply a [rolling update](https://docs.docker.com/engine/swarm/swarm-tutorial/rolling-update/).

[tutorial](https://docs.docker.com/search/?q=tutorial), [cluster management](https://docs.docker.com/search/?q=cluster management), [swarm](https://docs.docker.com/search/?q=swarm), [service](https://docs.docker.com/search/?q=service)

# Apply rolling updates to a service

Estimated reading time: 4 minutes

In a previous step of the tutorial, you [scaled](https://docs.docker.com/engine/swarm/swarm-tutorial/scale-service/) the number of instances of a service. In this part of the tutorial, you deploy a service based on the Redis 3.0.6 container tag. Then you upgrade the service to use the Redis 3.0.7 container image using rolling updates.

1. If you haven’t already, open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named manager1.
2. Deploy your Redis tag to the swarm and configure the swarm with a 10 second update delay. Note that the following example shows an older Redis tag:

$ docker service create \

--replicas 3 \

--name redis \

--update-delay 10s \

redis:3.0.6

0u6a4s31ybk7yw2wyvtikmu50

You configure the rolling update policy at service deployment time.

The --update-delay flag configures the time delay between updates to a service task or sets of tasks. You can describe the time T as a combination of the number of seconds Ts, minutes Tm, or hours Th. So 10m30s indicates a 10 minute 30 second delay.

By default the scheduler updates 1 task at a time. You can pass the --update-parallelism flag to configure the maximum number of service tasks that the scheduler updates simultaneously.

By default, when an update to an individual task returns a state of RUNNING, the scheduler schedules another task to update until all tasks are updated. If, at any time during an update a task returns FAILED, the scheduler pauses the update. You can control the behavior using the --update-failure-action flag for docker service create or docker service update.

1. Inspect the redis service:

$ docker service inspect --pretty redis

ID: 0u6a4s31ybk7yw2wyvtikmu50

Name: redis

Service Mode: Replicated

Replicas: 3

Placement:

Strategy: Spread

UpdateConfig:

Parallelism: 1

Delay: 10s

ContainerSpec:

Image: redis:3.0.6

Resources:

Endpoint Mode: vip

1. Now you can update the container image for redis. The swarm manager applies the update to nodes according to the UpdateConfig policy:

$ docker service update --image redis:3.0.7 redis

redis

The scheduler applies rolling updates as follows by default:

* + Stop the first task.
  + Schedule update for the stopped task.
  + Start the container for the updated task.
  + If the update to a task returns RUNNING, wait for the specified delay period then start the next task.
  + If, at any time during the update, a task returns FAILED, pause the update.

1. Run docker service inspect --pretty redis to see the new image in the desired state:

$ docker service inspect --pretty redis

ID: 0u6a4s31ybk7yw2wyvtikmu50

Name: redis

Service Mode: Replicated

Replicas: 3

Placement:

Strategy: Spread

UpdateConfig:

Parallelism: 1

Delay: 10s

ContainerSpec:

Image: redis:3.0.7

Resources:

Endpoint Mode: vip

The output of service inspect shows if your update paused due to failure:

$ docker service inspect --pretty redis

ID: 0u6a4s31ybk7yw2wyvtikmu50

Name: redis

...snip...

Update status:

State: paused

Started: 11 seconds ago

Message: update paused due to failure or early termination of task 9p7ith557h8ndf0ui9s0q951b

...snip...

To restart a paused update run docker service update <SERVICE-ID>. For example:

$ docker service update redis

To avoid repeating certain update failures, you may need to reconfigure the service by passing flags to docker service update.

1. Run docker service ps <SERVICE-ID> to watch the rolling update:

$ docker service ps redis

NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR

redis.1.dos1zffgeofhagnve8w864fco redis:3.0.7 worker1 Running Running 37 seconds

\\_ redis.1.88rdo6pa52ki8oqx6dogf04fh redis:3.0.6 worker2 Shutdown Shutdown 56 seconds ago

redis.2.9l3i4j85517skba5o7tn5m8g0 redis:3.0.7 worker2 Running Running About a minute

\\_ redis.2.66k185wilg8ele7ntu8f6nj6i redis:3.0.6 worker1 Shutdown Shutdown 2 minutes ago

redis.3.egiuiqpzrdbxks3wxgn8qib1g redis:3.0.7 worker1 Running Running 48 seconds

\\_ redis.3.ctzktfddb2tepkr45qcmqln04 redis:3.0.6 mmanager1 Shutdown Shutdown 2 minutes ago

Before Swarm updates all of the tasks, you can see that some are running redis:3.0.6 while others are running redis:3.0.7. The output above shows the state once the rolling updates are done.

## What’s next?

Next, learn about how to [drain a node](https://docs.docker.com/engine/swarm/swarm-tutorial/drain-node/) in the swarm.[tutorial](https://docs.docker.com/search/?q=tutorial), [cluster management](https://docs.docker.com/search/?q=cluster management), [swarm](https://docs.docker.com/search/?q=swarm), [service](https://docs.docker.com/search/?q=service), [rolling-update](https://docs.docker.com/search/?q=rolling-update)

# Drain a node on the swarm

Estimated reading time: 3 minutes

In earlier steps of the tutorial, all the nodes have been running with ACTIVE availability. The swarm manager can assign tasks to any ACTIVE node, so up to now all nodes have been available to receive tasks.

Sometimes, such as planned maintenance times, you need to set a node to DRAIN availability. DRAIN availability prevents a node from receiving new tasks from the swarm manager. It also means the manager stops tasks running on the node and launches replica tasks on a node with ACTIVE availability.

**Important**: Setting a node to DRAIN does not remove standalone containers from that node, such as those created with docker run, docker-compose up, or the Docker Engine API. A node’s status, including DRAIN, only affects the node’s ability to schedule swarm service workloads.

1. If you haven’t already, open a terminal and ssh into the machine where you run your manager node. For example, the tutorial uses a machine named manager1.
2. Verify that all your nodes are actively available.

$ docker node ls

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS

1bcef6utixb0l0ca7gxuivsj0 worker2 Ready Active

38ciaotwjuritcdtn9npbnkuz worker1 Ready Active

e216jshn25ckzbvmwlnh5jr3g \* manager1 Ready Active Leader

1. If you aren’t still running the redis service from the [rolling update](https://docs.docker.com/engine/swarm/swarm-tutorial/rolling-update/) tutorial, start it now:

$ docker service create --replicas 3 --name redis --update-delay 10s redis:3.0.6

c5uo6kdmzpon37mgj9mwglcfw

1. Run docker service ps redis to see how the swarm manager assigned the tasks to different nodes:

$ docker service ps redis

NAME IMAGE NODE DESIRED STATE CURRENT STATE

redis.1.7q92v0nr1hcgts2amcjyqg3pq redis:3.0.6 manager1 Running Running 26 seconds

redis.2.7h2l8h3q3wqy5f66hlv9ddmi6 redis:3.0.6 worker1 Running Running 26 seconds

redis.3.9bg7cezvedmkgg6c8yzvbhwsd redis:3.0.6 worker2 Running Running 26 seconds

In this case the swarm manager distributed one task to each node. You may see the tasks distributed differently among the nodes in your environment.

1. Run docker node update --availability drain <NODE-ID> to drain a node that had a task assigned to it:

$ docker node update --availability drain worker1

worker1

1. Inspect the node to check its availability:

$ docker node inspect --pretty worker1

ID: 38ciaotwjuritcdtn9npbnkuz

Hostname: worker1

Status:

State: Ready

Availability: Drain

...snip...

The drained node shows Drain for AVAILABILITY.

1. Run docker service ps redis to see how the swarm manager updated the task assignments for the redis service:

$ docker service ps redis

NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR

redis.1.7q92v0nr1hcgts2amcjyqg3pq redis:3.0.6 manager1 Running Running 4 minutes

redis.2.b4hovzed7id8irg1to42egue8 redis:3.0.6 worker2 Running Running About a minute

\\_ redis.2.7h2l8h3q3wqy5f66hlv9ddmi6 redis:3.0.6 worker1 Shutdown Shutdown 2 minutes ago

redis.3.9bg7cezvedmkgg6c8yzvbhwsd redis:3.0.6 worker2 Running Running 4 minutes

The swarm manager maintains the desired state by ending the task on a node with Drain availability and creating a new task on a node with Active availability.

1. Run docker node update --availability active <NODE-ID> to return the drained node to an active state:

$ docker node update --availability active worker1

worker1

1. Inspect the node to see the updated state:

$ docker node inspect --pretty worker1

ID: 38ciaotwjuritcdtn9npbnkuz

Hostname: worker1

Status:

State: Ready

Availability: Active

...snip...

When you set the node back to Active availability, it can receive new tasks:

* + during a service update to scale up
  + during a rolling update
  + when you set another node to Drain availability
  + when a task fails on another active node

# Use swarm mode routing mesh

Estimated reading time: 8 minutes

Docker Engine swarm mode makes it easy to publish ports for services to make them available to resources outside the swarm. All nodes participate in an ingress **routing mesh**. The routing mesh enables each node in the swarm to accept connections on published ports for any service running in the swarm, even if there’s no task running on the node. The routing mesh routes all incoming requests to published ports on available nodes to an active container.

To use the ingress network in the swarm, you need to have the following ports open between the swarm nodes before you enable swarm mode:

* Port 7946 TCP/UDP for container network discovery.
* Port 4789 UDP for the container ingress network.

You must also open the published port between the swarm nodes and any external resources, such as an external load balancer, that require access to the port.

You can also [bypass the routing mesh](https://docs.docker.com/engine/swarm/ingress/" \l "bypass-the-routing-mesh) for a given service.

## Publish a port for a service

Use the --publish flag to publish a port when you create a service. target is used to specify the port inside the container, and published is used to specify the port to bind on the routing mesh. If you leave off the published port, a random high-numbered port is bound for each service task. You need to inspect the task to determine the port.

$ docker service create \

--name <SERVICE-NAME> \

--publish published=<PUBLISHED-PORT>,target=<CONTAINER-PORT> \

<IMAGE>

**Note**: The older form of this syntax is a colon-separated string, where the published port is first and the target port is second, such as -p 8080:80. The new syntax is preferred because it is easier to read and allows more flexibility.

The <PUBLISHED-PORT> is the port where the swarm makes the service available. If you omit it, a random high-numbered port is bound. The <CONTAINER-PORT> is the port where the container listens. This parameter is required.

For example, the following command publishes port 80 in the nginx container to port 8080 for any node in the swarm:

$ docker service create \

--name my-web \

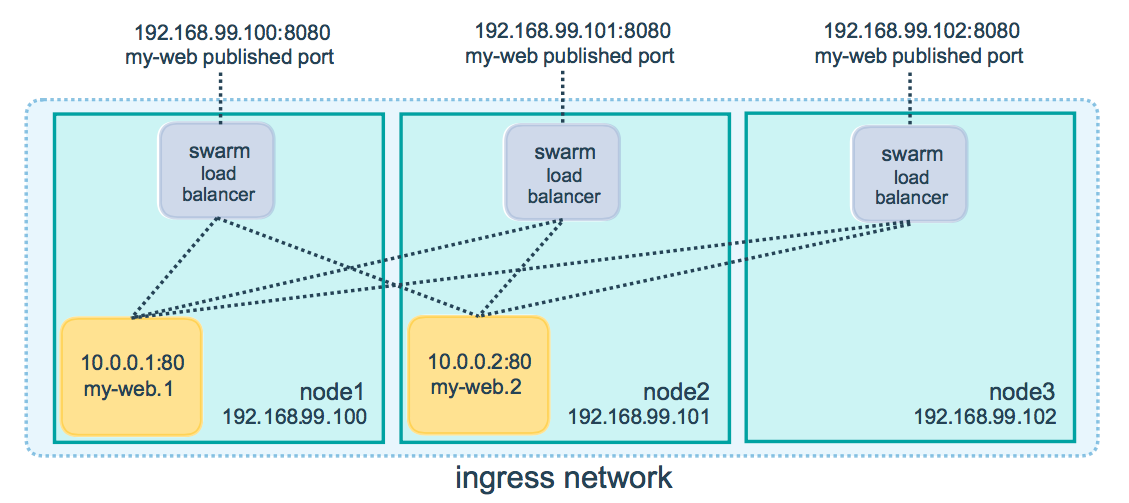
--publish published=8080,target=80 \

--replicas 2 \

nginx

When you access port 8080 on any node, Docker routes your request to an active container. On the swarm nodes themselves, port 8080 may not actually be bound, but the routing mesh knows how to route the traffic and prevents any port conflicts from happening.

The routing mesh listens on the published port for any IP address assigned to the node. For externally routable IP addresses, the port is available from outside the host. For all other IP addresses the access is only available from within the host.



You can publish a port for an existing service using the following command:

$ docker service update \

--publish-add published=<PUBLISHED-PORT>,target=<CONTAINER-PORT> \

<SERVICE>

You can use docker service inspect to view the service’s published port. For instance:

$ docker service inspect --format="{{json .Endpoint.Spec.Ports}}" my-web

[{"Protocol":"tcp","TargetPort":80,"PublishedPort":8080}]

The output shows the <CONTAINER-PORT> (labeled TargetPort) from the containers and the <PUBLISHED-PORT> (labeled PublishedPort) where nodes listen for requests for the service.

### Publish a port for TCP only or UDP only

By default, when you publish a port, it is a TCP port. You can specifically publish a UDP port instead of or in addition to a TCP port. When you publish both TCP and UDP ports, If you omit the protocol specifier, the port is published as a TCP port. If you use the longer syntax (recommended), set the protocol key to either tcp or udp.

#### TCP only

**Long syntax:**

$ docker service create --name dns-cache \

--publish published=53,target=53 \

dns-cache

**Short syntax:**

$ docker service create --name dns-cache \

-p 53:53 \

dns-cache

#### TCP and UDP

**Long syntax:**

$ docker service create --name dns-cache \

--publish published=53,target=53 \

--publish published=53,target=53,protocol=udp \

dns-cache

**Short syntax:**

$ docker service create --name dns-cache \

-p 53:53 \

-p 53:53/udp \

dns-cache

#### UDP only

**Long syntax:**

$ docker service create --name dns-cache \

--publish published=53,target=53,protocol=udp \

dns-cache

**Short syntax:**

$ docker service create --name dns-cache \

-p 53:53/udp \

dns-cache

## Bypass the routing mesh

You can bypass the routing mesh, so that when you access the bound port on a given node, you are always accessing the instance of the service running on that node. This is referred to as host mode. There are a few things to keep in mind.

* If you access a node which is not running a service task, the service does not listen on that port. It is possible that nothing is listening, or that a completely different application is listening.
* If you expect to run multiple service tasks on each node (such as when you have 5 nodes but run 10 replicas), you cannot specify a static target port. Either allow Docker to assign a random high-numbered port (by leaving off the published), or ensure that only a single instance of the service runs on a given node, by using a global service rather than a replicated one, or by using placement constraints.

To bypass the routing mesh, you must use the long --publish service and set mode to host. If you omit the mode key or set it to ingress, the routing mesh is used. The following command creates a global service using host mode and bypassing the routing mesh.

$ docker service create --name dns-cache \

--publish published=53,target=53,protocol=udp,mode=host \

--mode global \

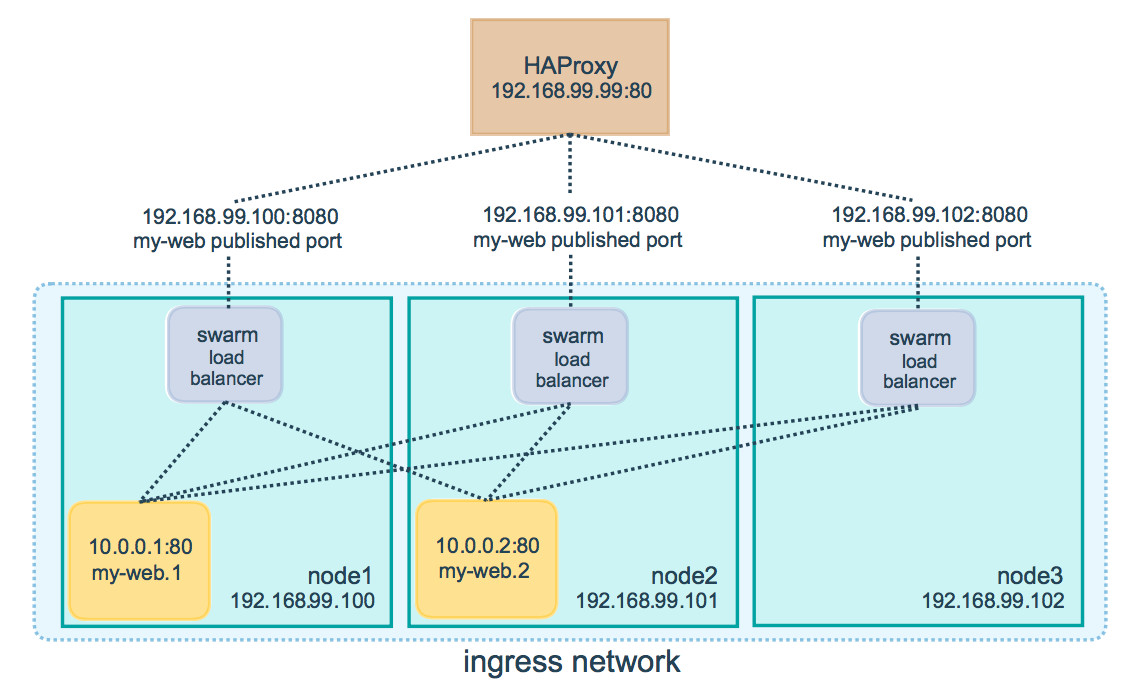
dns-cache

## Configure an external load balancer

You can configure an external load balancer for swarm services, either in combination with the routing mesh or without using the routing mesh at all.

### Using the routing mesh

You can configure an external load balancer to route requests to a swarm service. For example, you could configure [HAProxy](https://www.haproxy.org/) to balance requests to an nginx service published to port 8080.



In this case, port 8080 must be open between the load balancer and the nodes in the swarm. The swarm nodes can reside on a private network that is accessible to the proxy server, but that is not publicly accessible.

You can configure the load balancer to balance requests between every node in the swarm even if there are no tasks scheduled on the node. For example, you could have the following HAProxy configuration in /etc/haproxy/haproxy.cfg:

global

log /dev/log local0

log /dev/log local1 notice

...snip...

# Configure HAProxy to listen on port 80

frontend http\_front

bind \*:80

stats uri /haproxy?stats

default\_backend http\_back

# Configure HAProxy to route requests to swarm nodes on port 8080

backend http\_back

balance roundrobin

server node1 192.168.99.100:8080 check

server node2 192.168.99.101:8080 check

server node3 192.168.99.102:8080 check

When you access the HAProxy load balancer on port 80, it forwards requests to nodes in the swarm. The swarm routing mesh routes the request to an active task. If, for any reason the swarm scheduler dispatches tasks to different nodes, you don’t need to reconfigure the load balancer.

You can configure any type of load balancer to route requests to swarm nodes. To learn more about HAProxy, see the [HAProxy documentation](https://cbonte.github.io/haproxy-dconv/).

### Without the routing mesh

To use an external load balancer without the routing mesh, set --endpoint-mode to dnsrr instead of the default value of vip. In this case, there is not a single virtual IP. Instead, Docker sets up DNS entries for the service such that a DNS query for the service name returns a list of IP addresses, and the client connects directly to one of these. You are responsible for providing the list of IP addresses and ports to your load balancer. See [Configure service discovery](https://docs.docker.com/engine/swarm/networking/" \l "configure-service-discovery).

## Learn more

* [Deploy services to a swarm](https://docs.docker.com/engine/swarm/services/)

[guide](https://docs.docker.com/search/?q=guide), [swarm mode](https://docs.docker.com/search/?q=swarm mode), [swarm](https://docs.docker.com/search/?q=swarm), [network](https://docs.docker.com/search/?q=network), [ingress](https://docs.docker.com/search/?q=ingress), [routing mesh](https://docs.docker.com/search/?q=routing mesh)

# 10 Docker Security Best Practices

## Docker container security

The topic of Docker container security raises concerns ranging from Dockerfile security—relating to the Docker base images and potential security misconfigurations,—to the Docker container security at runtime regarding network ports, user privileges, Docker mounted filesystem access, and others. In this article, we will focus on the Docker container security aspects related to building a Docker image, reducing the security vulnerabilities count introduced by Docker base images as well as Dockerfile security best practices.

### **What is Docker security?**

Docker security refers to the build, runtime, and orchestration aspects of Docker containers. It includes the Dockerfile security aspects of Docker base images, as well as the Docker container security runtime aspects—such as user privileges, Docker daemon, proper CPU controls for a container, and further concerns around the orchestration of Docker containers at scale.

### THE STATE OF DOCKER CONTAINER SECURITY UNFOLDS INTO 4 MAIN DOCKER SECURITY ISSUES:

1. Dockerfile security and best practices
2. Docker container security at runtime
3. Supply chain security risks with Docker Hub and how they impact Docker container images
4. Cloud native container orchestration security aspects related to Kubernetes and Helm

In this installment of our cheat sheets, we’d like to focus on Docker security and discuss docker security best practices and guidelines that ensure a more secure and quality Docker images.

Also check out our [**Docker security report: Shifting Docker security left**](https://snyk.io/blog/shifting-docker-security-left/)

[](https://res.cloudinary.com/snyk/image/upload/v1551798390/Docker_Image_Security_Best_Practices_.pdf)

# Docker security

Estimated reading time: 11 minutes

There are four major areas to consider when reviewing Docker security:

* the intrinsic security of the kernel and its support for namespaces and cgroups;
* the attack surface of the Docker daemon itself;
* loopholes in the container configuration profile, either by default, or when customized by users.
* the “hardening” security features of the kernel and how they interact with containers.

## Kernel namespaces

Docker containers are very similar to LXC containers, and they have similar security features. When you start a container with docker run, behind the scenes Docker creates a set of namespaces and control groups for the container.

**Namespaces provide the first and most straightforward form of isolation**: processes running within a container cannot see, and even less affect, processes running in another container, or in the host system.

**Each container also gets its own network stack**, meaning that a container doesn’t get privileged access to the sockets or interfaces of another container. Of course, if the host system is setup accordingly, containers can interact with each other through their respective network interfaces — just like they can interact with external hosts. When you specify public ports for your containers or use [*links*](https://docs.docker.com/network/links/) then IP traffic is allowed between containers. They can ping each other, send/receive UDP packets, and establish TCP connections, but that can be restricted if necessary. From a network architecture point of view, all containers on a given Docker host are sitting on bridge interfaces. This means that they are just like physical machines connected through a common Ethernet switch; no more, no less.

How mature is the code providing kernel namespaces and private networking? Kernel namespaces were introduced [between kernel version 2.6.15 and 2.6.26](https://man7.org/linux/man-pages/man7/namespaces.7.html). This means that since July 2008 (date of the 2.6.26 release ), namespace code has been exercised and scrutinized on a large number of production systems. And there is more: the design and inspiration for the namespaces code are even older. Namespaces are actually an effort to reimplement the features of [OpenVZ](https://en.wikipedia.org/wiki/OpenVZ) in such a way that they could be merged within the mainstream kernel. And OpenVZ was initially released in 2005, so both the design and the implementation are pretty mature.

## Control groups

Control Groups are another key component of Linux Containers. They implement resource accounting and limiting. They provide many useful metrics, but they also help ensure that each container gets its fair share of memory, CPU, disk I/O; and, more importantly, that a single container cannot bring the system down by exhausting one of those resources.

So while they do not play a role in preventing one container from accessing or affecting the data and processes of another container, they are essential to fend off some denial-of-service attacks. They are particularly important on multi-tenant platforms, like public and private PaaS, to guarantee a consistent uptime (and performance) even when some applications start to misbehave.

Control Groups have been around for a while as well: the code was started in 2006, and initially merged in kernel 2.6.24.

## Docker daemon attack surface

Running containers (and applications) with Docker implies running the Docker daemon. This daemon requires root privileges unless you opt-in to [Rootless mode](https://docs.docker.com/engine/security/rootless/), and you should therefore be aware of some important details.

First of all, **only trusted users should be allowed to control your Docker daemon**. This is a direct consequence of some powerful Docker features. Specifically, Docker allows you to share a directory between the Docker host and a guest container; and it allows you to do so without limiting the access rights of the container. This means that you can start a container where the /host directory is the / directory on your host; and the container can alter your host filesystem without any restriction. This is similar to how virtualization systems allow filesystem resource sharing. Nothing prevents you from sharing your root filesystem (or even your root block device) with a virtual machine.

This has a strong security implication: for example, if you instrument Docker from a web server to provision containers through an API, you should be even more careful than usual with parameter checking, to make sure that a malicious user cannot pass crafted parameters causing Docker to create arbitrary containers.

For this reason, the REST API endpoint (used by the Docker CLI to communicate with the Docker daemon) changed in Docker 0.5.2, and now uses a UNIX socket instead of a TCP socket bound on 127.0.0.1 (the latter being prone to cross-site request forgery attacks if you happen to run Docker directly on your local machine, outside of a VM). You can then use traditional UNIX permission checks to limit access to the control socket.

You can also expose the REST API over HTTP if you explicitly decide to do so. However, if you do that, be aware of the above mentioned security implications. Note that even if you have a firewall to limit accesses to the REST API endpoint from other hosts in the network, the endpoint can be still accessible from containers, and it can easily result in the privilege escalation. Therefore it is mandatory to secure API endpoints with [HTTPS and certificates](https://docs.docker.com/engine/security/protect-access/). It is also recommended to ensure that it is reachable only from a trusted network or VPN.

You can also use DOCKER\_HOST=ssh://USER@HOST or ssh -L /path/to/docker.sock:/var/run/docker.sock instead if you prefer SSH over TLS.

The daemon is also potentially vulnerable to other inputs, such as image loading from either disk with docker load, or from the network with docker pull. As of Docker 1.3.2, images are now extracted in a chrooted subprocess on Linux/Unix platforms, being the first-step in a wider effort toward privilege separation. As of Docker 1.10.0, all images are stored and accessed by the cryptographic checksums of their contents, limiting the possibility of an attacker causing a collision with an existing image.

Finally, if you run Docker on a server, it is recommended to run exclusively Docker on the server, and move all other services within containers controlled by Docker. Of course, it is fine to keep your favorite admin tools (probably at least an SSH server), as well as existing monitoring/supervision processes, such as NRPE and collectd.

## Linux kernel capabilities

By default, Docker starts containers with a restricted set of capabilities. What does that mean?

Capabilities turn the binary “root/non-root” dichotomy into a fine-grained access control system. Processes (like web servers) that just need to bind on a port below 1024 do not need to run as root: they can just be granted the net\_bind\_service capability instead. And there are many other capabilities, for almost all the specific areas where root privileges are usually needed.

This means a lot for container security; let’s see why!

Typical servers run several processes as root, including the SSH daemon, cron daemon, logging daemons, kernel modules, network configuration tools, and more. A container is different, because almost all of those tasks are handled by the infrastructure around the container:

* SSH access are typically managed by a single server running on the Docker host;
* cron, when necessary, should run as a user process, dedicated and tailored for the app that needs its scheduling service, rather than as a platform-wide facility;
* log management is also typically handed to Docker, or to third-party services like Loggly or Splunk;
* hardware management is irrelevant, meaning that you never need to run udevd or equivalent daemons within containers;
* network management happens outside of the containers, enforcing separation of concerns as much as possible, meaning that a container should never need to perform ifconfig, route, or ip commands (except when a container is specifically engineered to behave like a router or firewall, of course).

This means that in most cases, containers do not need “real” root privileges at all. And therefore, containers can run with a reduced capability set; meaning that “root” within a container has much less privileges than the real “root”. For instance, it is possible to:

* deny all “mount” operations;
* deny access to raw sockets (to prevent packet spoofing);
* deny access to some filesystem operations, like creating new device nodes, changing the owner of files, or altering attributes (including the immutable flag);
* deny module loading;
* and many others.

This means that even if an intruder manages to escalate to root within a container, it is much harder to do serious damage, or to escalate to the host.

This doesn’t affect regular web apps, but reduces the vectors of attack by malicious users considerably. By default Docker drops all capabilities except [those needed](https://github.com/moby/moby/blob/master/oci/caps/defaults.go" \l "L6-L19), an allowlist instead of a denylist approach. You can see a full list of available capabilities in [Linux manpages](https://man7.org/linux/man-pages/man7/capabilities.7.html).

One primary risk with running Docker containers is that the default set of capabilities and mounts given to a container may provide incomplete isolation, either independently, or when used in combination with kernel vulnerabilities.

Docker supports the addition and removal of capabilities, allowing use of a non-default profile. This may make Docker more secure through capability removal, or less secure through the addition of capabilities. The best practice for users would be to remove all capabilities except those explicitly required for their processes.

## Docker Content Trust Signature Verification

The Docker Engine can be configured to only run signed images. The Docker Content Trust signature verification feature is built directly into the dockerd binary.  
This is configured in the Dockerd configuration file.

To enable this feature, trustpinning can be configured in daemon.json, whereby only repositories signed with a user-specified root key can be pulled and run.

This feature provides more insight to administrators than previously available with the CLI for enforcing and performing image signature verification.

For more information on configuring Docker Content Trust Signature Verificiation, go to [Content trust in Docker](https://docs.docker.com/engine/security/trust/).

## Other kernel security features

Capabilities are just one of the many security features provided by modern Linux kernels. It is also possible to leverage existing, well-known systems like TOMOYO, AppArmor, SELinux, GRSEC, etc. with Docker.

While Docker currently only enables capabilities, it doesn’t interfere with the other systems. This means that there are many different ways to harden a Docker host. Here are a few examples.

* You can run a kernel with GRSEC and PAX. This adds many safety checks, both at compile-time and run-time; it also defeats many exploits, thanks to techniques like address randomization. It doesn’t require Docker-specific configuration, since those security features apply system-wide, independent of containers.
* If your distribution comes with security model templates for Docker containers, you can use them out of the box. For instance, we ship a template that works with AppArmor and Red Hat comes with SELinux policies for Docker. These templates provide an extra safety net (even though it overlaps greatly with capabilities).
* You can define your own policies using your favorite access control mechanism.

Just as you can use third-party tools to augment Docker containers, including special network topologies or shared filesystems, tools exist to harden Docker containers without the need to modify Docker itself.

As of Docker 1.10 User Namespaces are supported directly by the docker daemon. This feature allows for the root user in a container to be mapped to a non uid-0 user outside the container, which can help to mitigate the risks of container breakout. This facility is available but not enabled by default.

Refer to the [daemon command](https://docs.docker.com/engine/reference/commandline/dockerd/" \l "daemon-user-namespace-options) in the command line reference for more information on this feature. Additional information on the implementation of User Namespaces in Docker can be found in [this blog post](https://integratedcode.us/2015/10/13/user-namespaces-have-arrived-in-docker/).

## Conclusions

Docker containers are, by default, quite secure; especially if you run your processes as non-privileged users inside the container.

You can add an extra layer of safety by enabling AppArmor, SELinux, GRSEC, or another appropriate hardening system.

If you think of ways to make docker more secure, we welcome feature requests, pull requests, or comments on the Docker community forums.

## Related information

* [Use trusted images](https://docs.docker.com/engine/security/trust/)
* [Seccomp security profiles for Docker](https://docs.docker.com/engine/security/seccomp/)
* [AppArmor security profiles for Docker](https://docs.docker.com/engine/security/apparmor/)
* [On the Security of Containers (2014)](https://medium.com/@ewindisch/on-the-security-of-containers-2c60ffe25a9e)
* [Docker swarm mode overlay network security model](https://docs.docker.com/network/overlay/)

[Docker](https://docs.docker.com/search/?q=Docker), [Docker documentation](https://docs.docker.com/search/?q=Docker documentation), [security](https://docs.docker.com/search/?q=security)