Docker frequently asked questions (FAQ)

Estimated reading time: 9 minutes

Does Docker run on Linux, macOS, and Windows?

You can run both Linux and Windows programs and executables in Docker containers. The Docker platform runs natively on Linux (on x86-64, ARM and many other CPU architectures) and on Windows (x86-64).

Docker Inc. builds products that let you build and run containers on Linux, Windows and macOS.

What does Docker technology add to just plain LXC?

Docker technology is not a replacement for LXC. "LXC" refers to capabilities of the Linux kernel (specifically namespaces and control groups) which allow sandboxing processes from one another, and controlling their resource allocations. On top of this low-level foundation of kernel features, Docker offers a high-level tool with several powerful functionalities:

- *Portable deployment across machines.* Docker defines a format for bundling an application and all its dependencies into a single object called a container. This container can be transferred to any Docker-enabled machine. The container can be executed there with the guarantee that the execution environment exposed to the application is the same in development, testing, and production. LXC implements process sandboxing, which is an important pre-requisite for portable deployment, but is not sufficient for portable deployment. If you sent me a copy of your application installed in a custom LXC configuration, it would almost certainly not run on my machine the way it does on yours. The app you sent me is tied to your machine's specific configuration: networking, storage, logging, etc. Docker defines an abstraction for these machine-specific settings. The exact same Docker container can run - unchanged - on many different machines, with many different configurations.
- *Application-centric.* Docker is optimized for the deployment of applications, as opposed to machines. This is reflected in its API, user interface, design philosophy and documentation. By contrast, the `lxc` helper scripts focus on containers as lightweight machines - basically servers that boot faster and need less RAM. We think there's more to containers than just that.
- *Automatic build.* Docker includes *a tool for developers to automatically assemble a container from their source code*, with full control over application dependencies, build tools, packaging etc. They are free to use `make`, `maven`, `chef`, `puppet`, `salt,` Debian packages, RPMs, source tarballs, or any combination of the above, regardless of the configuration of the machines.
- *Versioning.* Docker includes git-like capabilities for tracking successive versions of a container, inspecting the diff between versions, committing new versions, rolling back etc. The history also includes how a container was assembled and by whom, so you get full traceability from the production server all the way back to the upstream developer. Docker also implements incremental uploads and downloads, similar to `git pull`, so new versions of a container can be transferred by only sending diffs.
- *Component re-use.* Any container can be used as a *parent image* to create more specialized components. This can be done manually or as part of an automated build. For example you can prepare the ideal Python environment, and use it as a base for 10 different applications. Your ideal PostgreSQL setup can be re-used for all your future projects. And so on.
- *Sharing.* Docker has access to a public registry on Docker Hub where thousands of people have uploaded useful images: anything from Redis, CouchDB, PostgreSQL to IRC bouncers to Rails app servers to Hadoop to base images for various Linux distros. The *registry* also includes an official "standard library" of useful containers maintained by the Docker team. The registry itself is open-source, so anyone can deploy their own registry to store and transfer private containers, for internal server deployments for example.
- *Tool ecosystem.* Docker defines an API for automating and customizing the creation and deployment of containers. There are a huge number of tools integrating with Docker to extend its capabilities. PaaS-like deployment (Dokku, Deis, Flynn), multi-node orchestration (Maestro, Salt, Mesos, Openstack Nova), management dashboards (docker-ui, Openstack Horizon, Shipyard), configuration management (Chef, Puppet), continuous integration (Jenkins, Strider, Travis), etc. Docker is rapidly establishing itself as the standard for container-based tooling.

What is different between a Docker container and a VM?

There's a great StackOverflow answer showing the differences.

Do I lose my data when the container exits?

Not at all! Any data that your application writes to disk gets preserved in its container until you explicitly delete the container. The file system for the container persists even after the container halts.

How far do Docker containers scale?

Some of the largest server farms in the world today are based on containers. Large web deployments like Google and Twitter, and platform providers such as Heroku run on container technology, at a scale of hundreds of thousands or even millions of containers.

How do I connect Docker containers?

Currently the recommended way to connect containers is via the Docker network feature. You can see details of how to work with Docker networks.

How do I run more than one process in a Docker container?

This approach is discouraged for most use cases. For maximum efficiency and isolation, each container should address one specific area of concern. However, if you need to run multiple services within a single container, see Run multiple services in a container.

How do I report a security issue with Docker?

You can learn about the project's security policy here and report security issues to this mailbox.

Why do I need to sign my commits to Docker with the DCO?

Read our blog post on the introduction of the DCO.

When building an image, should I prefer system libraries or bundled ones?

This is a summary of a discussion on the docker-dev mailing list.

Virtually all programs depend on third-party libraries. Most frequently, they use dynamic linking and some kind of package dependency, so that when multiple programs need the same library, it is installed only once.

Some programs, however, bundle their third-party libraries, because they rely on very specific versions of those libraries.

When creating a Docker image, is it better to use the bundled libraries, or should you build those programs so that they use the default system libraries instead?

The key point about system libraries is not about saving disk or memory space. It is about security. All major distributions handle security seriously, by having dedicated security teams, following up closely with published vulnerabilities, and disclosing advisories themselves. (Look at the Debian Security Information for an example of those procedures.) Upstream developers, however, do not always implement similar practices.

Before setting up a Docker image to compile a program from source, if you want to use bundled libraries, you should check if the upstream authors provide a convenient way to announce security vulnerabilities, and if they update their bundled libraries in a timely manner. If they don't, you are exposing yourself (and the users of your image) to security vulnerabilities.

Likewise, before using packages built by others, you should check if the channels providing those packages implement similar security best practices. Downloading and installing an "all-in-one" .deb or .rpm sounds great at first, except if you have no way to figure out that it contains a copy of the OpenSSL library vulnerable to the Heartbleed bug.

Why is `DEBIAN_FRONTEND=noninteractive` discouraged in Dockerfiles?

When building Docker images on Debian and Ubuntu you may have seen errors like:

```
unable to initialize frontend: Dialog
```

These errors don't stop the image from being built but inform you that the installation process tried to open a dialog box, but couldn't. Generally, these errors are safe to ignore.

Some people circumvent these errors by changing the `DEBIAN_FRONTEND` environment variable inside the Dockerfile using:

```
ENV DEBIAN_FRONTEND=noninteractive
```

This prevents the installer from opening dialog boxes during installation which stops the errors.

While this may sound like a good idea, it *may* have side effects. The `DEBIAN_FRONTEND` environment variable is inherited by all images and containers built from your image, effectively changing their behavior. People using those images run into problems when installing software interactively, because installers do not show any dialog boxes.

Because of this, and because setting `DEBIAN_FRONTEND` to `noninteractive` is mainly a 'cosmetic' change, we *discourage* changing it.

If you *really* need to change its setting, make sure to change it back to its default value afterwards.

Why do I get `Connection reset by peer` when making a request to a service running in a container?

Typically, this message is returned if the service is already bound to your localhost. As a result, requests coming to the container from outside are dropped. To correct this problem, change the service's configuration on your localhost so that the service accepts requests from all IPs. If you aren't sure how to do this, check the documentation for your OS.

Why do I get `Cannot connect to the Docker daemon. Is the docker daemon running on this host?` when using docker-machine?

This error points out that the docker client cannot connect to the virtual machine. This means that either the virtual machine that works underneath `docker-machine` is not running or that the client doesn't correctly point at it.

To verify that the docker machine is running you can use the `docker-machine ls` command and start it with `docker-machine start` if needed.

```
$ docker-machine ls
NAME             ACTIVE   DRIVER       STATE     URL   SWARM                 DOCKER    ERRORS
default          -        virtualbox   Stopped                                 Unknown

$ docker-machine start default
```

You need to tell Docker to talk to that machine. You can do this with the `docker-machine env` command. For example,

```
$ eval "$(docker-machine env default)"
$ docker ps
```

Where can I find more answers?

You can find more answers on:

- Docker community Slack channel
- Docker Support Forums
- GitHub
- Ask questions on Stackoverflow
- Join the conversation on Twitter

faq, questions, documentation, docker