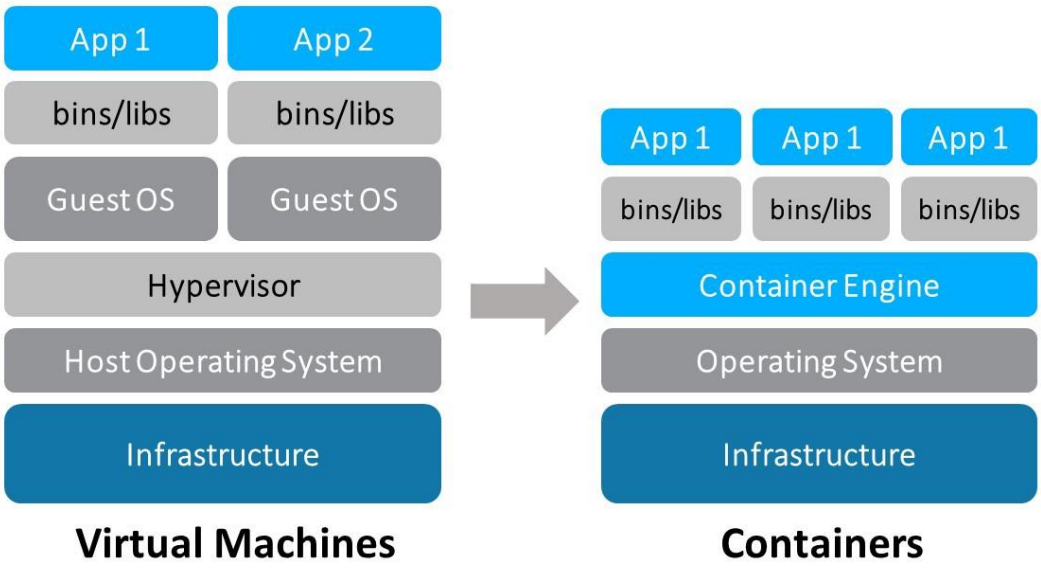
Docker basics

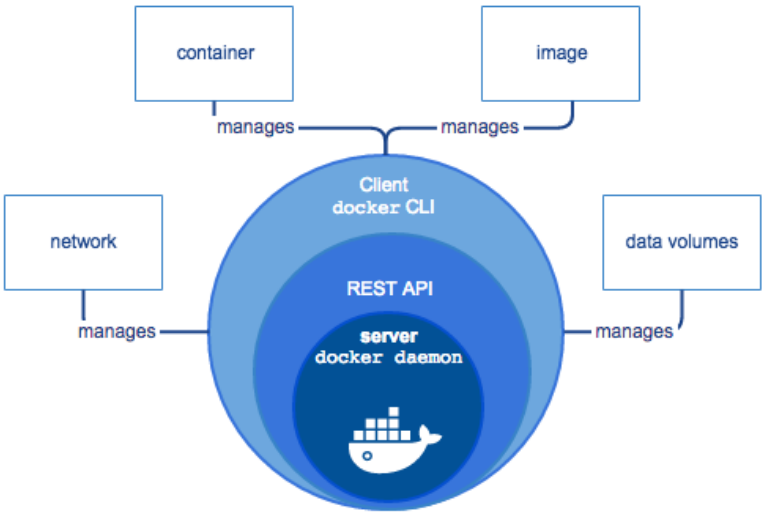
Docker is a computer program that performs operating-system-level virtualization, also known as "containerization"

A Docker container, unlike a virtual machine, does not require or include a separate operating system. Instead, it relies on the kernel's functionality and ***uses resource isolation for CPU and memory, and separate namespaces*** *to isolate the application's view of the operating system*

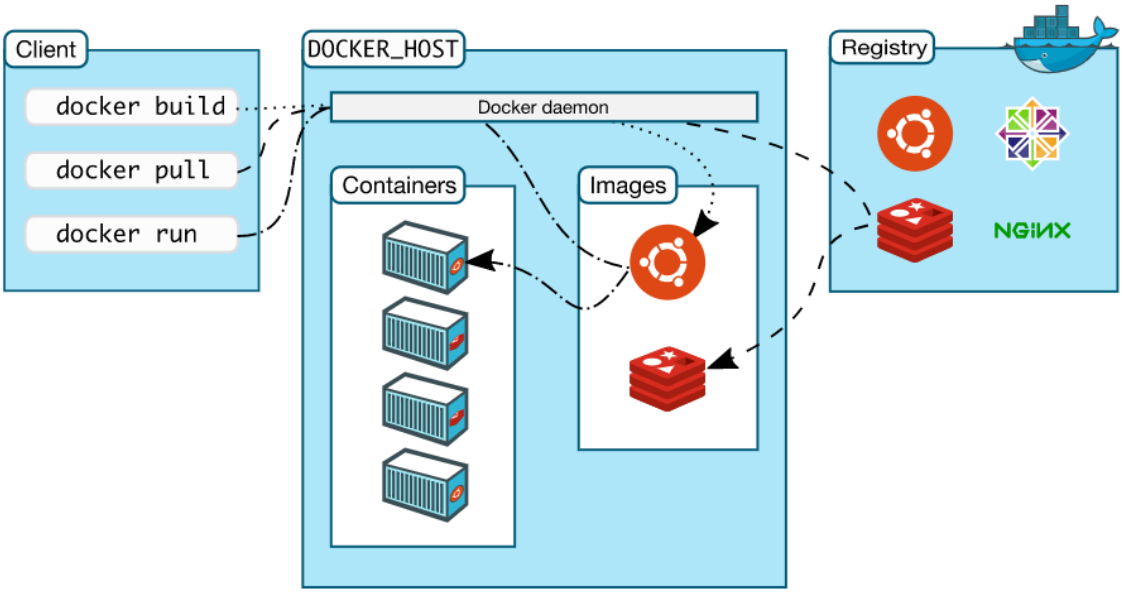


Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications.

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight because they don't need the extra load of a hypervisor, but run directly within the host machine's kernel. This means you can run more containers on a given hardware combination than if you were using virtual machines. You can even run Docker containers within host machines that are actually virtual machines!



- 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.



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.

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 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. If you use Docker Datacenter (DDC), it includes Docker Trusted Registry (DTR).

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.

Building Java containers with Plugins

When it comes to Maven-Docker integration, there is no shortage of Maven plugins: spotify/docker-maven-plugin (now inactive), spotify/dockerfile-maven, JIB, boost-maven-plugin (Spring Boot-focused), fabric8io/docker-maven-plugin.

JIB

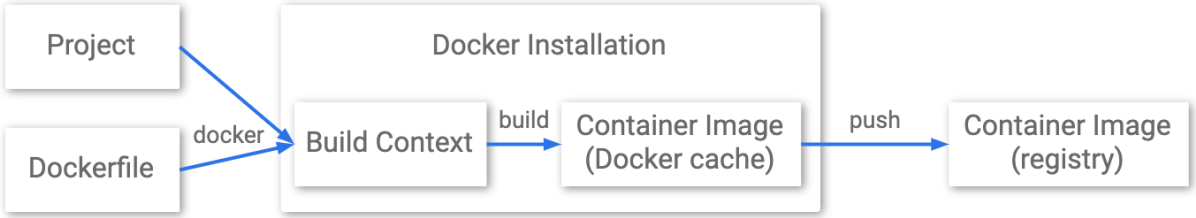
Jib builds containers without using a Dockerfile or requiring a Docker installation. You can use Jib in the Jib plugins for Maven or Gradle, or you can use the Jib Java library.

Jib is a fast and simple container image builder that handles all the steps of packaging your application into a container image. It does not require you to write a Dockerfile or have docker installed, and it is directly integrated into Maven and Gradle — just add the plugin to your build and you'll have your Java application containerized in no time.

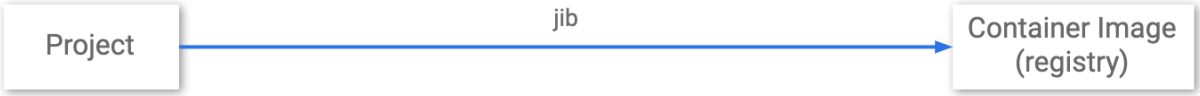
What does Jib do?

Jib handles all steps of packaging your application into a container image. You don't need to know best practices for creating Dockerfiles or have Docker installed.

Docker build flow:



Jib build flow:



Jib organizes your application into distinct layers; dependencies, resources, and classes; and utilizes Docker image layer caching to keep builds fast by only rebuilding changes. Jib's layer organization and small base image keeps overall image size small which improves performance and portability.

JIB will look into your pom file, figure out what you are trying to build, make some default sensible guesses, and produce a Docker image by directly building the layers of that image.

That effectively means that in CI/CD environments, where access to a Docker daemon may be problematic due to permissions (or quite challenging in Docker-in0Docker scenarios), JIB can

build you an image and save you the trouble. At this point you may be wondering, “So JIB is what I need, why should I bother with any other Docker-Maven plugins from now on?”

Well, JIB is powerful, flexible, and smart but has an important caveat: You can not execute RUN commands. You know, those `apt-get install`, `wget foo.tar`, `tar xfz bar.tar.gz`, etc. commands we usually see at the beginning of a Dockerfile?

The reason is simple and has to do with the very architecture of JIB. Since JIB does not utilize a Docker daemon, there is no temporary container created in which such RUN commands can be executed into. So this is a functionality that can probably never be supported in JIB unless its architecture is radically changed.

Fabric8 and Spotify

Traditionally, you work with Docker images by authoring a Dockerfile, and with the help of `docker build` and `docker push`, you build and push your image to a remote registry.

Fabric8 and Spotify plugins create docker image on the base of Dockerfile and Docker Daemon.

Pros: support of all Dockerfile functionality inclusive commands

Cons: Docker daemon installation is necessary on build server