Chapter 5 - Docker - Container Tool

Using Docker and am very impressed with how easily you can set up an environment to develop, test and deploy applications. It provides a virtualization environment, but unlike some of the other virtualization solutions, Docker container (runtime environment for your application) is a process instead of a complete OS. It runs natively on Linux and uses a single virtual machine on Windows and Mac in which Docker containers run.

Using Docker you can easily create containers required for your application to run, for example containers for database, application server etc and easily deploy them to test or production environments. You can create images of your set-up and re-create the same setup constantly from those images. When you create a Docker image, it has to be based on some variant of Linux base image. You can browse Docker images at Docker Hub.

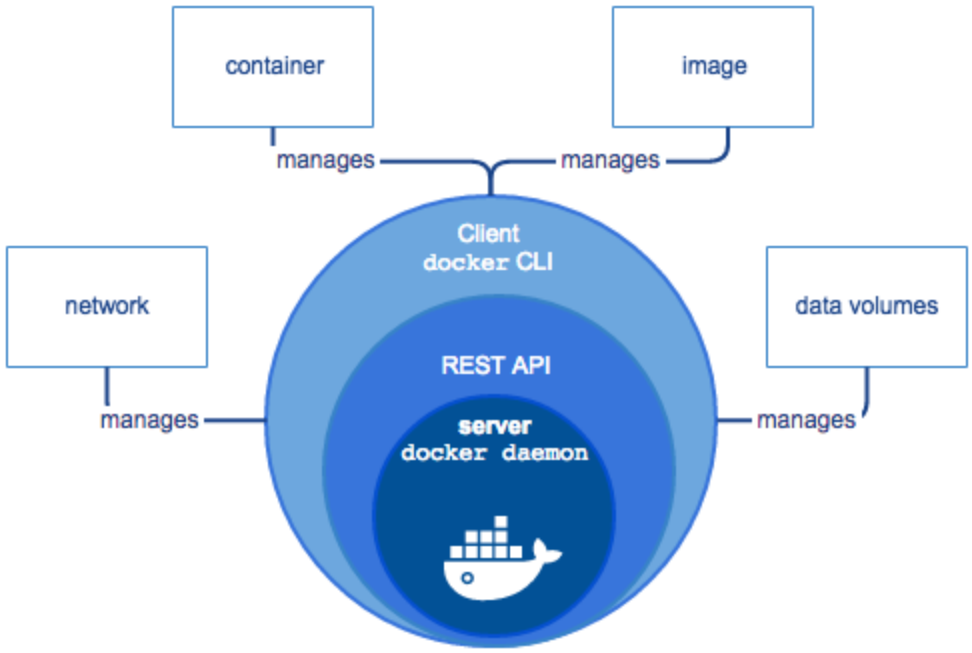
What is Docker?

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.

Docker Engine

Docker Engine is a *client-server* application with these major components:

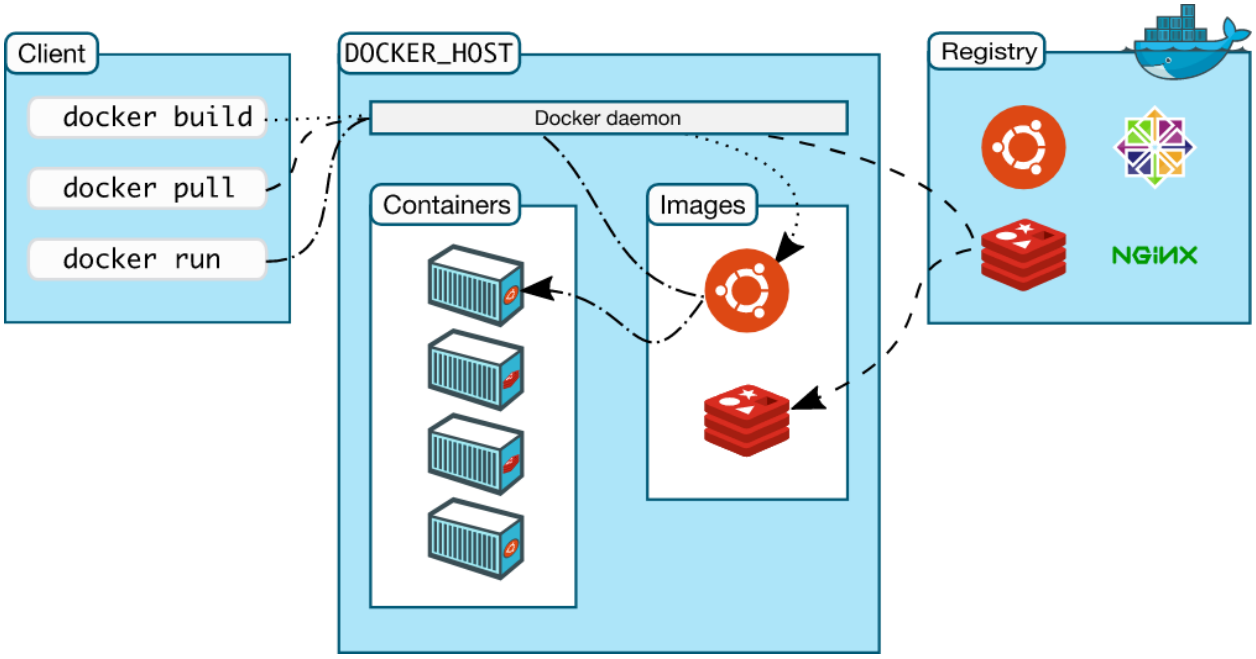
- A server which is a type of long-running program called a daemon process (the dockerd command).
- A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface (CLI) client (the docker command).



Docker Architecture

As previously mentioned, 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.



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.

- Image:
  - An *image* is a lightweight, stand-alone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files.
  - 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.
  - You might create your own images. To do that, 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.
- Container:
  - A *container* is a runtime instance of an image.
  - Isolation: It runs completely isolated from the host environment by default, only accessing host files and ports if configured to do so. 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.
  - Containers run apps natively on the host machine's kernel.

    - They have better performance characteristics than virtual machines that only get virtual access to host resources through a hypervisor.
    - Containers can get native access, each one running in a discrete process, taking no more memory than any other executable.

The underlying technology

Docker is written in Go and takes advantage of several features of the Linux kernel to deliver its functionality.

- Namespaces

Docker uses a technology called namespaces to provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container.

These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

Docker Engine uses namespaces such as the following on Linux:

- The pid namespace: Process isolation (PID: Process ID).
- The net namespace: Managing network interfaces (NET: Networking).
- The ipc namespace: Managing access to IPC resources (IPC: InterProcess Communication).
- The mnt namespace: Managing filesystem mount points (MNT: Mount).
- The uts namespace: Isolating kernel and version identifiers. (UTS: Unix Timesharing System).
- Control Groups

Docker Engine on Linux also relies on another technology called control groups (cgroups). A cgroup limits an application to a specific set of resources. Control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints. For example, you can limit the memory available to a specific container.

- Union File Systems

Union file systems, or UnionFS, are file systems that operate by creating layers, making them very lightweight and fast. Docker Engine uses UnionFS to provide the building blocks for containers. Docker Engine can use multiple UnionFS variants, including AUFS, btrfs, vfs, and DeviceMapper.

- Container Format

Docker Engine combines the namespaces, control groups, and UnionFS into a wrapper called a container format. The default container format is libcontainer. In the future, Docker may support other container formats by integrating with technologies such as BSD Jails or Solaris Zones.

Dockerfile

You create Dockerfile to create a new docker container. Some of the commands used in the Dockerfile are (also see complete official reference docs) –

FROM – Specify base image for your docker image. For example – FROM ubuntu

ADD – Add file(s) from the host machine to a docker container. For example, to copy setup.sh file, from the directory from where docker commands are run, to my container – ADD ./setup.sh /setup.sh

RUN – Runs a command in the container. For example to make setup.sh file executable after copying to the container – RUN chmod +x /setup.sh

ENTRYPOINT– Docker containers are meant to have one main application, which when stops running, the container stops. That main program is specified using this directive. For example to run Apache server after it is installed (using possibly RUN command) – ENTRYPOINT apachectl start

CMD – This is a little confusing directive in Dockerfile. There can be only one command specified in a Dockerfile. In the absence of ENTRYPOINT, if you specify CMD, with is executable, then that becomes the main application for the container. If you do not specify an executable application command in CMD, then arguments to it are passed to the command specified in ENTRYPOINT.

EXPOSE – This is another one of the confusing directives. You would think this command exposes ports from the container to the outside world. But that is not what it does. It simply tells Docker that the container listens on the specified port(s) at runtime. For example if Apache server is listening on port 80 in a container, then you would specify – EXPOSE 80 in the docker file

ENV – Set environment variable(s) in the container. For example, ENV PATH=/some/path:$PATH

VOLUME – Creates a mountable point for a volume. Volume is just like a folder, or virtual folder. From within the container, it can be accessed as any other folder. Volumes can be used to share folders across different running containers. One container can import volumes from another container.
There are many other directives that can be used in a Dockerfile, but above are the ones I have used frequently, well other than cmd.

Creating Docker images and containers

You can create a docker container either completely from command line options or passing the location of Dockerfile.

Create docker container with command line options

(See complete reference)

Create a container from ubuntu image

$ docker run -name my-ubuntu -it ubuntu bash

The above command will create a docker container from base ubuntu image, name it my-ubuntu, run bash command (open bash shell) and keep standard input open (-i) and text input console open (-t, together -it). It will open a bash shell in the container, where you can execute any command.

Create an image from Dockerfile

$ docker build -t image_name .

Run above command from the directory where Dockerfile is present. -t option specifies the name of the image in : format. See build command reference for details.

See currently running containers

$ docker ps

If you run this command from another terminal on the host computer (from where you are running docker commands), you will see my-ubuntu container created above. If you did not exit the bash shell opened in that container (exit command), then you should see my-ubuntu container listed. If you exit the shell, the container stops (because bash was the main command and it terminated). To see all containers, including stopped ones, use -a flag

$ docker ps -a

ps command shows a lot of information. However you can filter and format the output. Format should be a Go template string. For example to see only names of container use following command –

$ docker ps --format "{{.Names}}"

See docs for more formatting options.

Start Container

$ docker start my-ubuntu

Above command will start my-ubuntu container, if it was stopped. It will use the same entrypoint command that was specified when the container was created. In this case it will open bash shell, but will terminate immediately. So the container would also stop. Specifying -i option will keep stdin open and will allow you to run commands in the container.

$ docker start -i my-ubuntu

use stop command to stop the container, e.g. docker stop my-ubuntu1

To remove a container , use rm command (you can specify multiple containers names) –

$ docker rm my-ubuntu1 my-ubuntu

If you want to remove the running container, use -f option, e.g. docker rm -f my-ubuntu1. Instead of container names, you can use container ids also.

Attaching to running container

Let's create a container in detached mode.

$ docker run -d -it --name my-ubuntu ubuntu

-d option runs docker container in background (detached mode). You will immediately return to the command prompt after executing the above command.

To attach to the the above container and the process that started it (in this case /bin/bash) –

$ docker attach my-ubuntu

This will allow you to execute commands in the bash shell that was started in the container when the container was run. Existing the shell (exit command) will also terminate the container.

If you do not want to terminate the container upon existing the bash shell, you can use exec command. It can be used to run any command, not just bash shell.

$ docker exec -it my-ubuntu bash

This will open a new bash shell. Exiting that shell will not terminate the container because it was not the command that started the container.

Listing images

To list all images –

```
$ docker images
```

Deleting Image

To remove images, use rmi command. Note that there should be no container based on the images you want to delete. If there are containers using images to be deleted, then remove those containers first using the rm command mentioned above.

```
$ docker rmi my-image1 my-image2
```

Instead of names you can also use image ids.

Delete all Containers

Following command will delete ALL containers, so be careful

```
$ docker rm $(docker ps -a -q)
```

-q option tells ps command to return only ids, which are then fed to rm command.

Here is an example of using filters to remove containers (this example removes all containers starting with my-ubuntu)

```
$ docker rm $(docker ps --filter name=my-ubuntu* -q)
```

Delete all Images

Following command deletes all images, so again be careful –

```
$ docker rmi $(docker images -q)
```

To delete by filtering on image name –

```
$ docker rmi ($docker images *my-ubuntu*)
```

Using Volumes

As mentioned earlier, volumes can be used to share data between host and container and also amongst containers. Let's create a container, call it container1, with volume v-container1.

```
$ docker run -dt --name container1 -v /v-container1 ubuntu
```

You can open a bash shell in the container and verify that /v-container1 folder is available. Create a file in this folder.

```
$ docker exec -it container1 bash
$ touch v-container1/file-in-container1
```

Now create a container, container2, that uses volumes from container1.

```
$ docker run --rm -it --name container2 --volumes-from container1 ubuntu
```

Note that the above command uses –rm to create a temporary container. Once the main application started in the container (in this case bash shell) stops, the container will be terminated and it won't appear in 'docker ps -a' command too.

Once the shell opens in container2, you can ensure that /v-container1 is available and the file we created exists in the folder.

Mapping folder from host to container

To share a folder from the host to a container, use the same -v option, but specify <host-folder-name>:<path-in-container> argument. For example if you want make, say /project1/src folder from host map to /src folder in the container –

```
$ docker run --rm -it -v ${PWD}:/src ubuntu
```

${PWD} tells docker to map the present working directory. We could have entered the entire path too – docker run –rm -it -v /project1/src:/src ubuntu

If your host is Mac or Windows, make sure /project1/src folder is shared in Docker preferences, else docker will throw an error.

Using volumes for backup and restore

I found docker volumes very useful when backing up data from one container and restoring it in another. For example, if you had created a container from mysql image and populated a database, you could easily create a tar file of that data from the host machine and then restore it when you create a new instance of the container, or restore the data in the original db container. However you will need to know volumes used by the container – in this case the container using mysql base image. You can find that easily by running this command –

```
$ docker inspect my-db
```

Note that the output is JSON. Look for the "Mounts" key and "destination" sub-key in that. In the case of a mysql container, this value is "/var/lib/mysql". This is where the data is stored in a mysql container. So we can use the -volumes-from option we saw above to use this volume in another container and then run tar command on it and save it in the volume mapped on the host machine.

```
$ docker run --rm --volumes-from my-db -v ${pwd}/backup-data:/backup-data ubuntu tar cvf /backup-data/my-db-volume.tar /var/lib/mysql
```

We are using –rm because we want to create a temporary container. The container will be terminated after the command is finished. We are using volumes from my-db container, which is based on mysql image. This makes /var/lib/mysql folder available to this (temporary) container. Then we are mapping the backup-data folder in the present folder (on the host machine) to /backup-data in the container. So now the temporary container has access to /var/lib/mysql (from my-db container) and backup-data folder on the host machine. Then we execute tar command to create my-db-volume.tar in /backup-data, which in effect is created in the same named folder in the host machine. And it tars data from /var/lib/mysql, which contains data from my-db container.

To restore the data –

```
$ docker run --rm --volumes-from my-new-db -v $(pwd)/data-backup:/backup-data ubuntu bash -c "cd / && tar xvf /backup-data/my-db-data.tar"
```

Here we are restoring the data into a newly created my-new-db container (created with mysql base image). We are using volumes from the new db container, so /var/lib/mysql folder is available to the temporary container. We map the same folder from host to the temp container and then run tar command to untar /backup-data/my-db-data.tar (which **untars** the same named file from the host machine) into root. So data from /var/lib/mysql in the tar file will be written to /var/lib/mysql in the temp container. And since this volume is coming from my-new-db (the container into which we want to restore the data), we get data saved in that container.

Creating image from container

Let's say you create a container from some base image, you install more applications in that container. Now you want to save the container as an image so that next time you could create a container from that image and don't have to repeat installation of all the additional applications. You can do that with export command –

```
$ docker export -o /my-images/container1-image.tar container1
```

Specify output file path using -o option. The last argument is the name of the container from which you want to create an image.

To create image from the exported file, use import command –

```
$ docker import /my-images/container1-image.tar container1-image
```

The above command will create an image named container1-image from container1-image.tar file.

I know the post is getting really long, but I wanted to put some of the most useful commands, according to me, of Docker in this post. I would wrap up the post by briefly list some of the useful commands in docker-compose

Docker-Compose

Docker-compose can be used to configure multiple docker containers in the same file and also to specify dependencies between them. There is a docker-compose command which executes commands from docker-compose-yml (though you can override file name in docker-compose command).

Structure of docker-compose.yml is simple – It has version number at the top (the latest version is 3, but I have mostly worked with version 2), and then there is a services section which lists container definitions. Here an example of docker-compose.yml

```
version: "2"

services:
  my-db:
    image: mysql
    environment:
      MYSQL_DATABASE: my-db
    container_name: my-db
    ports:
      - '3307:3306'

  my-app-server:
    build: ../my-app-server
    container_name: my-app-server
    stdin_open: true
```

```
    tty: true
    ports:
      - '8082:8080'
      - '9001:9001' # Open port for debugging
    volumes:
      - ../src/app1:/src/app1
      - ../logs:/logs
    depends_on:
      - my-db
```

Many of the options we have already seen in this post so far, so I though I would just show and example of docker-compose.yml. The file contains definition of one db container that specifies image directly in the docker-compose.yml and one app server container that builds the container from dockerfile in ../my-app-server folder. Note the dependency of app server on db server, specified in 'depends_on' section in the app server

To bring up both the container, run

```
$ docker-compose up
```

To run docker-compose in detached mode, use -d option. However you may want to use non-detached mode to see output messages.

To stop all containers started in docker-compose.yml, press CTRL+C if it is running in foreground, or you can run

```
$ docker-compose down
```

Volumes

It is possible to store data within the writable layer of a container, but there are some downsides:

- The data won't persist when that container is no longer running, and it can be difficult to get the data out of the container if another process needs it.
- A container's writable layer is tightly coupled to the host machine where the container is running. You can't easily move the data somewhere else.
- Writing into a container's writable layer requires a storage driver to manage the filesystem. The storage driver provides a union filesystem, using the Linux kernel. This extra abstraction reduces performance as compared to using data volumes, which write directly to the host filesystem.

Docker offers three different ways to mount data into a container from the Docker host:

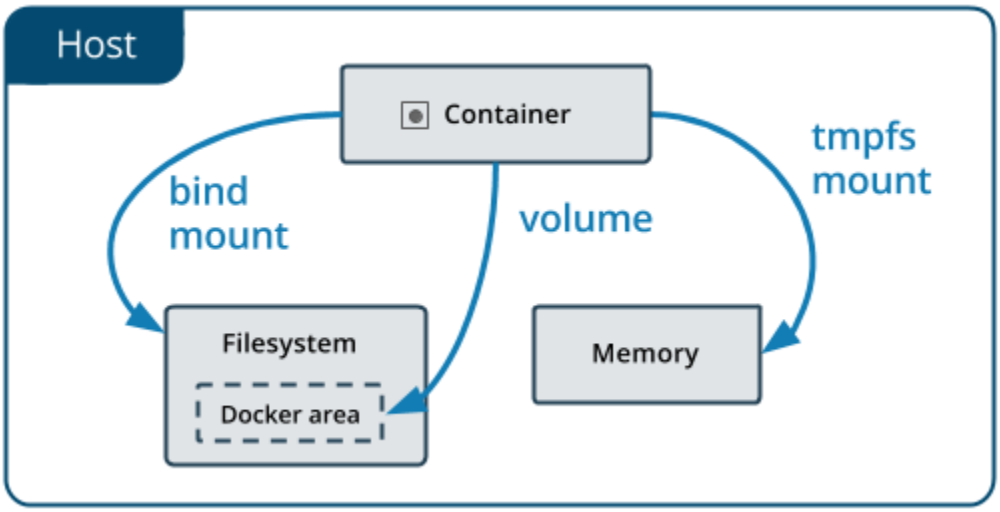
- Volumes
- Bind mounts
- Tmpfs volumes

When in doubt, volumes are almost always the right choice.

Choose the right type of mount

No matter which type of mount you choose to use, the data looks the same from within the container. It is exposed as either a directory or an individual file in the container's filesystem.

An easy way to visualize the difference among volumes, bind mounts, and tmpfs mounts is to think about where the data lives on the Docker host.



- **Volumes**
  Volumes are stored in a part of the host filesystem which is managed by Docker (/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.
  Volumes are created and managed by Docker. You can create a volume explicitly using the docker volume create command, or Docker can create a volume during container or service creation.
  When you create a volume, it is stored within a directory on the Docker host. When you mount the volume into a container, this directory is what is mounted into the container. This is similar to the way that bind mounts work, except that volumes are managed by Docker and are isolated from the core functionality of the host machine.
  A given volume can be mounted into multiple containers simultaneously. When no running container is using a volume, the volume is still available to Docker and is not removed automatically. You can remove unused volumes using docker volume prune.
  When you mount a volume, it may be named or anonymous. Anonymous volumes are not given an explicit name when they are first mounted into a container, so Docker gives them a random name that is guaranteed to be unique within a given Docker host.
- **Bind mounts**
  Bind mounts may be stored anywhere on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.
  Bind mounts have limited functionality compared to volumes. When you use a bind mount, a file or directory on the host machine is mounted into a container. The file or directory is referenced by its full path on the host machine.
  The file or directory does not need to exist on the Docker host already. It is created on demand if it does not yet exist. Bind mounts are very performant, but they rely on the host machine's filesystem having a specific directory structure available.
  If you are developing new Docker applications, consider using named volumes.
  Warning: One side effect of using bind mounts, for better or for worse, is that you can change the host filesystem via processes running in a container, including creating, modifying, or deleting important system files or directories. This is a powerful ability which can have security implications, including impacting non-Docker processes on the host system.
- **tmpfs mounts**
  tmpfs mounts are stored in the host system's memory only, and are never written to the host system's filesystem.
  A tmpfs mount is not persisted on disk, either on the Docker host or within a container. It can be used by a container during the lifetime of the container, to store non-persistent state or sensitive information. For instance, internally, swarm services use tmpfs mounts to mount secrets into a service's containers.

Good use cases for Volumes

Volumes are the preferred way to persist data in Docker containers and services. Some use cases for volumes include:

- Sharing data among multiple running containers. If you don't explicitly create it, a volume is created the first time it is mounted into a container. When that container stops or is removed, the volume still exists. Multiple containers can mount the same volume simultaneously, either read-write or read-only. Volumes are only removed when you explicitly remove them.
- When the Docker host is not guaranteed to have a given directory or file structure. Volumes help you decouple the configuration of the Docker host from the container runtime.
- When you want to store your container's data on a remote host or a cloud provider, rather than locally.
- When you need to be able to back up, restore, or migrate data from one Docker host to another, volumes are a better choice. You can stop containers using the volume, then back up the volume's directory (such as /var/lib/docker/volumes/).

Good use cases for Bind Mounts

In general, you should use volumes where possible. Bind mounts are appropriate for the following types of use case:

- Sharing configuration files from the host machine to containers. This is how Docker provides DNS resolution to containers by default, by mounting /etc/resolv.conf from the host machine into each container.
- Sharing source code or build artifacts between a development environment on the Docker host and a container. For instance, you may mount a Maven target/ directory into a container, and each time you build the Maven project on the Docker host, the container gets access to the rebuilt artifacts.

If you use Docker for development this way, your production Dockerfile would copy the production-ready artifacts directly into the image, rather than relying on a bind mount.

- When the file or directory structure of the Docker host is guaranteed to be consistent with the bind mounts the containers require.