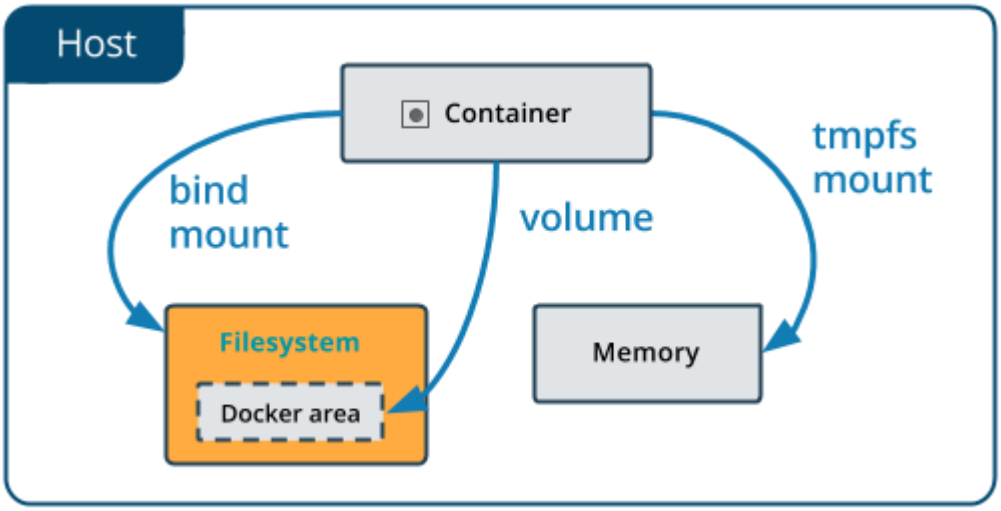
Using bind mounts

Bind mounts have been around since the early days of Docker. 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 or relative path on the host machine. By contrast, when you use a volume, a new directory is created within Docker's storage directory on the host machine, and Docker manages that directory's contents.

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 instead. You can't use Docker CLI commands to directly manage bind mounts.



Choose the -v or --mount flag

Originally, the `-v` or `--volume` flag was used for standalone containers and the `--mount` flag was used for swarm services. However, starting with Docker 17.06, you can also use `--mount` with standalone containers. 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.

> **Tip**: New users should use the `--mount` syntax. Experienced users may be more familiar with the `-v` or `--volume` syntax, but are encouraged to use `--mount`, because research has shown it to be easier to use.

- **`-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 bind mounts, the first field is the path to the file or directory on the **host machine**.
    - The second field is the path where the file or directory is mounted in the container.
    - The third field is optional, and is a comma-separated list of options, such as `ro`, `consistent`, `delegated`, `cached`, `z`, and `Z`. 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`, `volume`, or `tmpfs`. This topic discusses bind mounts, so the type is always `bind`.
- The `source` of the mount. For bind mounts, this is the path to the file or directory on the Docker daemon host. 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.
- The `bind-propagation` option, if present, changes the bind propagation. May be one of `rprivate`, `private`, `rshared`, `shared`, `rslave`, `slave`.
- The `consistency` option, if present, may be one of `consistent`, `delegated`, or `cached`. This setting only applies to Docker Desktop for Mac, and is ignored on all other platforms.
- The `--mount` flag does not support `z` or `Z` options for modifying selinux labels.

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

Differences between `-v` and `--mount` behavior

Because the `-v` and `--volume` flags have been a part of Docker for a long time, their behavior cannot be changed. This means that **there is one behavior that is different between `-v` and `--mount`.**

If you use `-v` or `--volume` to bind-mount a file or directory that does not yet exist on the Docker host, `-v` creates the endpoint for you. **It is always created as a directory.**

If you use `--mount` to bind-mount a file or directory that does not yet exist on the Docker host, Docker does **not** automatically create it for you, but generates an error.

Start a container with a bind mount

Consider a case where you have a directory `source` and that when you build the source code, the artifacts are saved into another directory, `source/target/`. You want the artifacts to be available to the container at `/app/`, and you want the container to get access to a new build each time you build the source on your development host. Use the following command to bind-mount the `target/` directory into your container at `/app/`. Run the command from within the `source` directory. The `$(pwd)` sub-command expands to the current working directory on Linux or macOS hosts.

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

- mount
- -v

- mount

```
$ docker run -d \
  -it \
  --name devtest \
```

```
  --mount type=bind,source="$(pwd)"/target,target=/app \
  nginx:latest
```

- -v

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app \
  nginx:latest
```

Use `docker inspect devtest` to verify that the bind mount was created correctly. Look for the `Mounts` section:

```
"Mounts": [
    {
        "Type": "bind",
        "Source": "/tmp/source/target",
        "Destination": "/app",
        "Mode": "",
        "RW": true,
        "Propagation": "rprivate"
    }
],
```

This shows that the mount is a `bind` mount, it shows the correct source and destination, it shows that the mount is read-write, and that the propagation is set to `rprivate`.

Stop the container:

```
$ docker container stop devtest

$ docker container rm devtest
```

Mount into a non-empty directory on the container

If you bind-mount into a non-empty directory on the container, the directory's existing contents are obscured by the bind mount. This can be beneficial, such as when you want to test a new version of your application without building a new image. However, it can also be surprising and this behavior differs from that of docker volumes.

This example is contrived to be extreme, but replaces the contents of the container's `/usr/` directory with the `/tmp/` directory on the host machine. In most cases, this would result in a non-functioning container.

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

- mount
- -v

- mount

```
$ docker run -d \
  -it \
  --name broken-container \
  --mount type=bind,source=/tmp,target=/usr \
  nginx:latest

docker: Error response from daemon: oci runtime error: container_linux.go:262:
starting container process caused "exec: \"nginx\": executable file not found in
$PATH".
```

- -v

```
$ docker run -d \
  -it \
  --name broken-container \
  -v /tmp:/usr \
  nginx:latest

docker: Error response from daemon: oci runtime error: container_linux.go:262:
starting container process caused "exec: \"nginx\": executable file not found in
$PATH".
```

The container is created but does not start. Remove it:

```
$ docker container rm broken-container
```

Use a read-only bind mount

For some development applications, the container needs to write into the bind mount, so changes are propagated back to the Docker host. At other times, the container only needs read access.

This example modifies the one above but mounts the directory as a read-only bind mount, 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

- mount

```
$ docker run -d \
  -it \
  --name devtest \
  --mount type=bind,source="$(pwd)"/target,target=/app,readonly \
  nginx:latest
```

- -v

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app:ro \
  nginx:latest
```

Use `docker inspect devtest` to verify that the bind mount was created correctly. Look for the `Mounts` section:

```
"Mounts": [
    {
        "Type": "bind",
        "Source": "/tmp/source/target",
        "Destination": "/app",
        "Mode": "ro",
        "RW": false,
        "Propagation": "rprivate"
    }
],
```

Stop the container:

```
$ docker container stop devtest

$ docker container rm devtest
```

Configure bind propagation

Bind propagation defaults to `rprivate` for both bind mounts and volumes. It is only configurable for bind mounts, and only on Linux host machines. Bind propagation is an advanced topic and many users never need to configure it.

Bind propagation refers to whether or not mounts created within a given bind-mount or named volume can be propagated to replicas of that mount. Consider a mount point `/mnt`, which is also mounted on `/tmp`. The propagation settings control whether a mount on `/tmp/a` would also be available on `/mnt/a`. Each propagation setting has a recursive counterpoint. In the case of recursion, consider that `/tmp/a` is also mounted as `/foo`. The propagation settings control whether `/mnt/a` and/or `/tmp/a` would exist.

Propagation setting	Description
`shared`	Sub-mounts of the original mount are exposed to replica mounts, and sub-mounts of replica mounts are also propagated to the original mount.
`slave`	similar to a shared mount, but only in one direction. If the original mount exposes a sub-mount, the replica mount can see it. However, if the replica mount exposes a sub-mount, the original mount cannot see it.
`private`	The mount is private. Sub-mounts within it are not exposed to replica mounts, and sub-mounts of replica mounts are not exposed to the original mount.
`rshared`	The same as shared, but the propagation also extends to and from mount points nested within any of the original or replica mount points.
`rslave`	The same as slave, but the propagation also extends to and from mount points nested within any of the original or replica mount points.
`rprivate`	The default. The same as private, meaning that no mount points anywhere within the original or replica mount points propagate in either direction.

Before you can set bind propagation on a mount point, the host filesystem needs to already support bind propagation.

For more information about bind propagation, see the Linux kernel documentation for shared subtree{: target="*blank*" *class*=""}.

The following example mounts the `target/` directory into the container twice, and the second mount sets both the `ro` option and the `rslave` bind propagation option.

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

- mount
- -v

- mount

```
$ docker run -d \
  -it \
  --name devtest \
```

```
  --mount type=bind,source="$(pwd)"/target,target=/app \
  --mount type=bind,source="$(pwd)"/target,target=/app2,readonly,bind-
propagation=rslave \
  nginx:latest
```

- -v

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app \
  -v "$(pwd)"/target:/app2:ro,rslave \
  nginx:latest
```

Now if you create `/app/foo/`, `/app2/foo/` also exists.

Configure the selinux label

If you use `selinux` you can add the `z` or `Z` options to modify the selinux label of the **host file or directory** being mounted into the container. This affects the file or directory on the host machine itself and can have consequences outside of the scope of Docker.

- The `z` option indicates that the bind mount content is shared among multiple containers.
- The `Z` option indicates that the bind mount content is private and unshared.

Use **extreme** caution with these options. Bind-mounting a system directory such as `/home` or `/usr` with the `Z` option renders your host machine inoperable and you may need to relabel the host machine files by hand.

> **Important**: When using bind mounts with services, selinux labels (`:Z` and `:z`), as well as `:ro` are ignored. See moby/moby #32579 for details. {:.important}

This example sets the `z` option to specify that multiple containers can share the bind mount's contents:

It is not possible to modify the selinux label using the `--mount` flag.

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app:z \
  nginx:latest
```

Configure mount consistency for macOS

Docker Desktop for Mac uses `osxfs` to propagate directories and files shared from macOS to the Linux VM. This propagation makes these directories and files available to Docker containers running on Docker Desktop

for Mac.

By default, these shares are fully-consistent, meaning that every time a write happens on the macOS host or through a mount in a container, the changes are flushed to disk so that all participants in the share have a fully-consistent view. Full consistency can severely impact performance in some cases. Docker 17.05 and higher introduce options to tune the consistency setting on a per-mount, per-container basis. The following options are available:

- `consistent` or `default`: The default setting with full consistency, as described above.
- `delegated`: The container runtime's view of the mount is authoritative. There may be delays before updates made in a container are visible on the host.
- `cached`: The macOS host's view of the mount is authoritative. There may be delays before updates made on the host are visible within a container.

These options are completely ignored on all host operating systems except macOS.

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

- mount
- -v

- mount

```
$ docker run -d \
  -it \
  --name devtest \
  --mount type=bind,source="$(pwd)"/target,destination=/app,consistency=cached \
  nginx:latest
```

- -v

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app:cached \
  nginx:latest
```
