Compose file version 3 reference

Estimated reading time: 75 minutes

Reference and guidelines

These topics describe version 3 of the Compose file format. This is the newest version.

Compose and Docker compatibility matrix

There are several versions of the Compose file format – 1, 2, 2.x, and 3.x. The table below is a quick look. For full details on what each version includes and how to upgrade, see **About versions and upgrading (https://docs.docker.com/compose/compose-file/compose-versioning/)**.

This table shows which Compose file versions support specific Docker releases.

Compose file format	Docker Engine release
3.7	18.06.0+
3.6	18.02.0+
3.5	17.12.0+
3.4	17.09.0+
3.3	17.06.0+
3.2	17.04.0+
3.1	1.13.1+
3.0	1.13.0+
2.4	17.12.0+
2.3	17.06.0+

Compose file format	Docker Engine release
2.2	1.13.0+
2.1	1.12.0+
2.0	1.10.0+
1.0	1.9.1.+

In addition to Compose file format versions shown in the table, the Compose itself is on a release schedule, as shown in Compose releases (https://github.com/docker/compose/releases/), but file format versions do not necessarily increment with each release. For example, Compose file format 3.0 was first introduced in Compose release 1.10.0 (https://github.com/docker/compose/releases/tag/1.10.0), and versioned gradually in subsequent releases.

Compose file structure and examples

Example Compose file version 3

The topics on this reference page are organized alphabetically by top-level key to reflect the structure of the Compose file itself. Top-level keys that define a section in the configuration file such as `build`, `deploy`, `depends_on`, `networks`, and so on, are listed with the options that support them as sub-topics. This maps to the `<key>: <option>: <value>` indent structure of the Compose file.

A good place to start is the Getting Started (https://docs.docker.com/get-started/) tutorial which uses version 3 Compose stack files to implement multi-container apps, service definitions, and swarm mode. Here are some Compose files used in the tutorial.

- Your first docker-compose.yml File (https://docs.docker.com/get-started/part3/#your-first-docker-composeyml-file)
- Add a new service and redeploy (https://docs.docker.com/get-started/part5/#add-a-new-service-and-redeploy)

Another good reference is the Compose file for the voting app sample used in the Docker for Beginners lab (https://github.com/docker/labs/tree/master/beginner/) topic on Deploying an app to a Swarm

(https://github.com/docker/labs/blob/master/beginner/chapters/votingapp.md). This is also shown on the accordion at the top of this section.

Service configuration reference

The Compose file is a YAML (http://yaml.org/) file defining services (/compose/compose-file/#service-configuration-reference), networks (/compose/compose-file/#network-configuration-reference) and volumes (/compose/compose-file/#volume-configuration-reference). The default path for a Compose file is `./docker-compose.yml`.

> **Tip**: You can use either a `.yml` or `.yaml` extension for this file. They both work.

A service definition contains configuration that is applied to each container started for that service, much like passing command-line parameters to `docker container create`. Likewise, network and volume definitions are analogous to `docker network create` and `docker volume create`.

As with `docker container create`, options specified in the Dockerfile, such as `CMD`, `EXPOSE`, `VOLUME`, `ENV`, are respected by default - you don't need to specify them again in `docker-compose.yml`.

You can use environment variables in configuration values with a Bash-like `${VARIABLE}` syntax - see variable substitution (/compose/compose-file/#variable-substitution) for full details.

This section contains a list of all configuration options supported by a service definition in version 3.

build

Configuration options that are applied at build time.

`build` can be specified either as a string containing a path to the build context:

```
version: "3.7"
services:
  webapp:
    build: ./dir
```

Or, as an object with the path specified under context (/compose/compose-file/#context) and optionally Dockerfile (/compose/compose-file/#dockerfile) and args (/compose/compose-file/#args):

```
version: "3.7"
services:
  webapp:
    build:
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        buildno: 1
```

If you specify `image` as well as `build`, then Compose names the built image with the `webapp` and optional `tag` specified in `image`:

```
build: ./dir
image: webapp:tag
```

This results in an image named `webapp` and tagged `tag`, built from `./dir`.

> **Note**: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file. The `docker stack` command accepts only pre-built images.

CONTEXT

Either a path to a directory containing a Dockerfile, or a url to a git repository.

When the value supplied is a relative path, it is interpreted as relative to the location of the Compose file. This directory is also the build context that is sent to the Docker daemon.

Compose builds and tags it with a generated name, and uses that image thereafter.

```
build:
  context: ./dir
```

DOCKERFILE

Alternate Dockerfile.

Compose uses an alternate file to build with. A build path must also be specified.

```
build:
  context: .
  dockerfile: Dockerfile-alternate
```

ARGS

Add build arguments, which are environment variables accessible only during the build process.

First, specify the arguments in your Dockerfile:

```
ARG buildno
ARG gitcommithash

RUN echo "Build number: $buildno"
RUN echo "Based on commit: $gitcommithash"
```

Then specify the arguments under the `build` key. You can pass a mapping or a list:

```
build:
  context: .
  args:
    buildno: 1
    gitcommithash: cdc3b19
```

```
build:
  context: .
  args:
    - buildno=1
    - gitcommithash=cdc3b19
```

> **Note**: In your Dockerfile, if you specify `ARG` before the `FROM` instruction, `ARG` is not available in the build instructions under `FROM`. If you need an argument to be available in both places, also specify it under the `FROM` instruction. See Understand how ARGS and FROM interact (https://docs.docker.com/engine/reference/builder/#understand-how-arg-and-from-interact) for usage details.

You can omit the value when specifying a build argument, in which case its value at build time is the value in the environment where Compose is running.

```
args:
  - buildno
  - gitcommithash
```

> **Note**: YAML boolean values (`true` , `false` , `yes` , `no` , `on` , `off`) must be enclosed in quotes, so that the parser interprets them as strings.

CACHE_FROM

> **Note**: This option is new in v3.2

A list of images that the engine uses for cache resolution.

```
build:
  context: .
  cache_from:
    - alpine:latest
    - corp/web_app:3.14
```

LABELS

> **Note**: This option is new in v3.3

Add metadata to the resulting image using Docker labels (https://docs.docker.com/engine/userguide/labels-custom-metadata/). You can use either an array or a dictionary.

We recommend that you use reverse-DNS notation to prevent your labels from conflicting with those used by other software.

```
build:
  context: .
  labels:
    com.example.description: "Accounting webapp"
    com.example.department: "Finance"
    com.example.label-with-empty-value: ""
```

```
build:
  context: .
  labels:
    - "com.example.description=Accounting webapp"
    - "com.example.department=Finance"
    - "com.example.label-with-empty-value"
```

SHM_SIZE

> Added in version 3.5 (https://docs.docker.com/compose/compose-file/compose-versioning/#version-35) file format

Set the size of the `/dev/shm` partition for this build's containers. Specify as an integer value representing the number of bytes or as a string expressing a byte value (/compose/compose-file/#specifying-byte-values).

```
build:
  context: .
  shm_size: '2gb'
```

```
build:
  context: .
  shm_size: 10000000
```

TARGET

> Added in version 3.4 (https://docs.docker.com/compose/compose-file/compose-versioning/#version-34) file format

Build the specified stage as defined inside the `Dockerfile` . See the multi-stage build docs (https://docs.docker.com/engine/userguide/eng-image/multistage-build/) for details.

```
build:
  context: .
  target: prod
```

cap_add, cap_drop

Add or drop container capabilities. See `man 7 capabilities` for a full list.

```
cap_add:
  - ALL

cap_drop:
  - NET_ADMIN
  - SYS_ADMIN
```

> Note: These options are ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

cgroup_parent

Specify an optional parent cgroup for the container.

```
cgroup_parent: m-executor-abcd
```

> Note: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

command

Override the default command.

```
command: bundle exec thin -p 3000
```

The command can also be a list, in a manner similar to dockerfile (https://docs.docker.com/engine/reference/builder/#cmd):

```
command: ["bundle", "exec", "thin", "-p", "3000"]
```

configs

Grant access to configs on a per-service basis using the per-service `configs` configuration. Two different syntax variants are supported.

> **Note**: The config must already exist or be defined in the top-level `configs` configuration (/compose/compose-file/#configs-configuration-reference) of this stack file, or stack deployment fails.

For more information on configs, see configs (https://docs.docker.com/engine/swarm/configs/).

SHORT SYNTAX

The short syntax variant only specifies the config name. This grants the container access to the config and mounts it at `/<config_name>` within the container. The source name and destination mountpoint are both set to the config name.

The following example uses the short syntax to grant the `redis` service access to the `my_config` and `my_other_config` configs. The value of `my_config` is set to the contents of the file `./my_config.txt`, and `my_other_config` is defined as an external resource, which means that it has already been defined in Docker, either by running the `docker config create` command or by another stack deployment. If the external config does not exist, the stack deployment fails with a `config not found` error.

> **Note**: `config` definitions are only supported in version 3.3 and higher of the compose file format.

```
version: "3.7"
services:
  redis:
    image: redis:latest
    deploy:
      replicas: 1
    configs:
      - my_config
      - my_other_config
configs:
  my_config:
    file: ./my_config.txt
  my_other_config:
    external: true
```

LONG SYNTAX

The long syntax provides more granularity in how the config is created within the service's task containers.

- `source` : The name of the config as it exists in Docker.
- `target` : The path and name of the file to be mounted in the service's task containers. Defaults to `/<source>` if not specified.
- `uid` and `gid` : The numeric UID or GID that owns the mounted config file within in the service's task containers. Both default to `0` on Linux if not specified. Not supported on Windows.
- `mode` : The permissions for the file that is mounted within the service's task containers, in octal notation. For instance, `0444` represents world-readable. The default is `0444` . Configs cannot be writable because they are mounted in a temporary filesystem, so if you set the writable bit, it is ignored. The executable bit can be set. If you aren't familiar with UNIX file permission modes, you may find this permissions calculator (http://permissions-calculator.org/) useful.

The following example sets the name of `my_config` to `redis_config` within the container, sets the mode to `0440` (group-readable) and sets the user and group to `103` . The `redis` service does not have access to the `my_other_config` config.

```
version: "3.7"
services:
  redis:
    image: redis:latest
    deploy:
      replicas: 1
    configs:
      - source: my_config
        target: /redis_config
        uid: '103'
        gid: '103'
        mode: 0440
configs:
  my_config:
    file: ./my_config.txt
  my_other_config:
    external: true
```

You can grant a service access to multiple configs and you can mix long and short syntax. Defining a config does not imply granting a service access to it.

container_name

Specify a custom container name, rather than a generated default name.

```
container_name: my-web-container
```

Because Docker container names must be unique, you cannot scale a service beyond 1 container if you have specified a custom name. Attempting to do so results in an error.

> **Note**: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

credential_spec

> **Note**: this option was added in v3.3.

Configure the credential spec for managed service account. This option is only used for services using Windows containers. The `credential_spec` must be in the format `file://<filename>` or `registry://<value-name>`.

When using `file:`, the referenced file must be present in the `CredentialSpecs` subdirectory in the Docker data directory, which defaults to `C:\ProgramData\Docker\` on Windows. The following example loads the credential spec from a file named `C:\ProgramData\Docker\CredentialSpecs\my-credential-spec.json`:

```
credential_spec:
  file: my-credential-spec.json
```

When using `registry:`, the credential spec is read from the Windows registry on the daemon's host. A registry value with the given name must be located in:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Virtualization\Containers\Cre
```

The following example load the credential spec from a value named `my-credential-spec` in the registry:

```
credential_spec:
  registry: my-credential-spec
```

depends_on

Express dependency between services, Service dependencies cause the following behaviors:

- `docker-compose up` starts services in dependency order. In the following example, `db` and `redis` are started before `web` .
- `docker-compose up SERVICE` automatically includes `SERVICE` 's dependencies. In the following example, `docker-compose up web` also creates and starts `db` and `redis` .
- `docker-compose stop` stops services in dependency order. In the following example, `web` is stopped before `db` and `redis` .

Simple example:

```
version: "3.7"
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```

> There are several things to be aware of when using `depends_on` :
>
> - `depends_on` does not wait for `db` and `redis` to be “ready” before starting `web` - only until they have been started. If you need to wait for a service to be ready, see Controlling startup order (https://docs.docker.com/compose/startup-order/) for more on this problem and strategies for solving it.
> - Version 3 no longer supports the `condition` form of `depends_on` .
> - The `depends_on` option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a version 3 Compose file.

deploy

> Version 3 (https://docs.docker.com/compose/compose-file/compose-versioning/#version-3) only.

Specify configuration related to the deployment and running of services. This only takes effect when deploying to a swarm (https://docs.docker.com/engine/swarm/) with docker stack deploy (https://docs.docker.com/engine/reference/commandline/stack_deploy/), and is ignored by `docker-compose up` and `docker-compose run`.

```
version: "3.7"
services:
  redis:
    image: redis:alpine
    deploy:
      replicas: 6
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
```

Several sub-options are available:

ENDPOINT_MODE

Specify a service discovery method for external clients connecting to a swarm.

> Version 3.3 (https://docs.docker.com/compose/compose-file/compose-versioning/#version-3) only.

- `endpoint_mode: vip` - Docker assigns the service a virtual IP (VIP) that acts as the front end for clients to reach the service on a network. Docker routes requests between the client and available worker nodes for the service, without client knowledge of how many nodes are participating in the service or their IP addresses or ports. (This is the default.)
- `endpoint_mode: dnsrr` - DNS round-robin (DNSRR) service discovery does not use a single virtual IP. 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. DNS round-robin is useful in cases where you want to use your own load balancer, or for Hybrid Windows and Linux applications.

```
version: "3.7"

services:
  wordpress:
    image: wordpress
    ports:
      - "8080:80"
    networks:
      - overlay
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: vip

  mysql:
    image: mysql
    volumes:
       - db-data:/var/lib/mysql/data
    networks:
       - overlay
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: dnsrr

volumes:
  db-data:

networks:
  overlay:
```

The options for `endpoint_mode` also work as flags on the swarm mode CLI command docker service create (https://docs.docker.com/engine/reference/commandline/service_create/). For a quick list of all swarm related `docker` commands, see Swarm mode CLI commands (https://docs.docker.com/engine/swarm/#swarm-mode-key-concepts-and-tutorial).

To learn more about service discovery and networking in swarm mode, see Configure service discovery (https://docs.docker.com/engine/swarm/networking/#configure-service-discovery) in the swarm mode topics.

LABELS

Specify labels for the service. These labels are *only* set on the service, and *not* on any containers for the service.

```
version: "3.7"
services:
  web:
    image: web
    deploy:
      labels:
        com.example.description: "This label will appear on the web service"
```

To set labels on containers instead, use the `labels` key outside of `deploy` :

```
version: "3.7"
services:
  web:
    image: web
    labels:
      com.example.description: "This label will appear on all containers for th
```

MODE

Either `global` (exactly one container per swarm node) or `replicated` (a specified number of containers). The default is `replicated` . (To learn more, see Replicated and global services (https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/#replicated-and-global-services) in the swarm (https://docs.docker.com/engine/swarm/) topics.)

```
version: "3.7"
services:
  worker:
    image: dockersamples/examplevotingapp_worker
    deploy:
      mode: global
```

PLACEMENT

Specify placement of constraints and preferences. See the docker service create documentation for a full description of the syntax and available types of constraints (https://docs.docker.com/engine/reference/commandline/service_create/#specify-service-constraints-constraint) and preferences (https://docs.docker.com/engine/reference/commandline/service_create/#specify-service-placement-preferences-placement-pref).

```
version: "3.7"
services:
  db:
    image: postgres
    deploy:
      placement:
        constraints:
          - node.role == manager
          - engine.labels.operatingsystem == ubuntu 14.04
        preferences:
          - spread: node.labels.zone
```

REPLICAS

If the service is `replicated` (which is the default), specify the number of containers that should be running at any given time.

```
version: "3.7"
services:
  worker:
    image: dockersamples/examplevotingapp_worker
    networks:
      - frontend
      - backend
    deploy:
      mode: replicated
      replicas: 6
```

RESOURCES

Configures resource constraints.

> **Note**: This replaces the older resource constraint options (https://docs.docker.com/compose/compose-file/compose-file-v2/#cpu-and-other-resources) for non swarm mode in Compose files prior to version 3 (`cpu_shares` , `cpu_quota` , `cpuset` , `mem_limit` , `memswap_limit` , `mem_swappiness`), as described in Upgrading version 2.x to 3.x (https://docs.docker.com/compose/compose-file/compose-versioning/#upgrading).

Each of these is a single value, analogous to its docker service create (https://docs.docker.com/engine/reference/commandline/service_create/) counterpart.

In this general example, the `redis` service is constrained to use no more than 50M of memory and `0.50` (50% of a single core) of available processing time (CPU), and has `20M` of memory and `0.25` CPU time reserved (as always available to it).

```
version: "3.7"
services:
  redis:
    image: redis:alpine
    deploy:
      resources:
        limits:
          cpus: '0.50'
          memory: 50M
        reservations:
          cpus: '0.25'
          memory: 20M
```

The topics below describe available options to set resource constraints on services or containers in a swarm.

> **Looking for options to set resources on non swarm mode containers?**
>
> The options described here are specific to the `deploy` key and swarm mode. If you want to set resource constraints on non swarm deployments, use Compose file format version 2 CPU, memory, and other resource options (https://docs.docker.com/compose/compose-file/compose-file-v2/#cpu-and-other-resources). If you have further questions, refer to the discussion on the GitHub issue docker/compose/4513 (https://github.com/docker/compose/issues/4513).

Out Of Memory Exceptions (OOME)

If your services or containers attempt to use more memory than the system has available, you may experience an Out Of Memory Exception (OOME) and a container, or the Docker daemon, might be killed by the kernel OOM killer. To prevent this from happening, ensure that your application runs on hosts with adequate memory and see Understand the risks of running out of memory (https://docs.docker.com/engine/admin/resource_constraints/#understand-the-risks-of-running-out-of-memory).

RESTART_POLICY

Configures if and how to restart containers when they exit. Replaces `restart` (https://docs.docker.com/compose/compose-file/compose-file-v2/#orig-resources).

- `condition` : One of `none` , `on-failure` or `any` (default: `any`).
- `delay` : How long to wait between restart attempts, specified as a duration (/compose/compose-file/#specifying-durations) (default: 0).
- `max_attempts` : How many times to attempt to restart a container before giving up (default: never give up). If the restart does not succeed within the configured `window` , this attempt doesn't count toward the configured `max_attempts` value. For example, if `max_attempts` is set to '2', and the restart fails on the first attempt, more than two restarts may be attempted.
- `window` : How long to wait before deciding if a restart has succeeded, specified as a duration (/compose/compose-file/#specifying-durations) (default: decide immediately).

```
version: "3.7"
services:
  redis:
    image: redis:alpine
    deploy:
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3
        window: 120s
```

ROLLBACK_CONFIG

> Version 3.7 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-37) and up

Configures how the service should be rollbacked in case of a failing update.

- `parallelism` : The number of containers to rollback at a time. If set to 0, all containers rollback simultaneously.
- `delay` : The time to wait between each container group's rollback (default 0s).
- `failure_action` : What to do if a rollback fails. One of `continue` or `pause` (default `pause`)
- `monitor` : Duration after each task update to monitor for failure `(ns|us|ms|s|m|h)` (default 0s).
- `max_failure_ratio` : Failure rate to tolerate during a rollback (default 0).
- `order` : Order of operations during rollbacks. One of `stop-first` (old task is stopped before starting new one), or `start-first` (new task is started first, and the running tasks briefly overlap) (default `stop-first`).

UPDATE_CONFIG

Configures how the service should be updated. Useful for configuring rolling updates.

- `parallelism` : The number of containers to update at a time.
- `delay` : The time to wait between updating a group of containers.
- `failure_action` : What to do if an update fails. One of `continue` , `rollback` , or `pause` (default: `pause`).
- `monitor` : Duration after each task update to monitor for failure `(ns|us|ms|s|m|h)` (default 0s).
- `max_failure_ratio` : Failure rate to tolerate during an update.
- `order` : Order of operations during updates. One of `stop-first` (old task is stopped before starting new one), or `start-first` (new task is started first, and the running tasks briefly overlap) (default `stop-first`) **Note**: Only supported for v3.4 and higher.

> **Note**: `order` is only supported for v3.4 and higher of the compose file format.

```
version: "3.7"
services:
  vote:
    image: dockersamples/examplevotingapp_vote:before
    depends_on:
      - redis
    deploy:
      replicas: 2
      update_config:
        parallelism: 2
        delay: 10s
        order: stop-first
```

NOT SUPPORTED FOR `DOCKER STACK DEPLOY`

The following sub-options (supported for `docker-compose up` and `docker-compose run`) are *not supported* for `docker stack deploy` or the `deploy` key.

- build (/compose/compose-file/#build)
- cgroup_parent (/compose/compose-file/#cgroup_parent)
- container_name (/compose/compose-file/#container_name)
- devices (/compose/compose-file/#devices)
- tmpfs (/compose/compose-file/#tmpfs)
- external_links (/compose/compose-file/#external_links)
- links (/compose/compose-file/#links)
- network_mode (/compose/compose-file/#network_mode)
- restart (/compose/compose-file/#restart)
- security_opt (/compose/compose-file/#security_opt)
- sysctls (/compose/compose-file/#sysctls)
- userns_mode (/compose/compose-file/#userns_mode)

> **Tip:** See the section on how to configure volumes for services, swarms, and docker-stack.yml files (/compose/compose-file/#volumes-for-services-swarms-and-stack-files). Volumes *are* supported but to work with swarms and services, they must be configured as named volumes or associated with services that are constrained to nodes with access to the requisite volumes.

devices

List of device mappings. Uses the same format as the `--device` docker client create option.

```
devices:
  - "/dev/ttyUSB0:/dev/ttyUSB0"
```

> **Note**: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

depends_on

Express dependency between services, Service dependencies cause the following behaviors:

- `docker-compose up` starts services in dependency order. In the following example, `db` and `redis` are started before `web` .
- `docker-compose up SERVICE` automatically includes `SERVICE` 's dependencies. In the following example, `docker-compose up web` also creates and starts `db` and `redis` .
- `docker-compose stop` stops services in dependency order. In the following example, `web` is stopped before `db` and `redis` .

Simple example:

```
version: "3.7"
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```

> **There are several things to be aware of when using `depends_on` :**
>
> - `depends_on` does not wait for `db` and `redis` to be "ready" before starting `web` - only until they have been started. If you need to wait for a service to be ready, see Controlling startup order (https://docs.docker.com/compose/startup-order/) for more on this problem and strategies for solving it.
> - Version 3 no longer supports the `condition` form of `depends_on` .
> - The `depends_on` option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a version 3 Compose file.

dns

Custom DNS servers. Can be a single value or a list.

```
dns: 8.8.8.8
```

```
dns:
  - 8.8.8.8
  - 9.9.9.9
```

dns_search

Custom DNS search domains. Can be a single value or a list.

```
dns_search: example.com
```

```
dns_search:
  - dc1.example.com
  - dc2.example.com
```

entrypoint

Override the default entrypoint.

```
entrypoint: /code/entrypoint.sh
```

The entrypoint can also be a list, in a manner similar to dockerfile (https://docs.docker.com/engine/reference/builder/#entrypoint):

```
entrypoint:
    - php
    - -d
    - zend_extension=/usr/local/lib/php/extensions/no-debug-non-zts-20100525/xc
    - -d
    - memory_limit=-1
    - vendor/bin/phpunit
```

> **Note**: Setting `entrypoint` both overrides any default entrypoint set on the service's image with the `ENTRYPOINT` Dockerfile instruction, *and* clears out any default command on the image - meaning that if there's a `CMD` instruction in the Dockerfile, it is ignored.

env_file

Add environment variables from a file. Can be a single value or a list.

If you have specified a Compose file with `docker-compose -f FILE`, paths in `env_file` are relative to the directory that file is in.

Environment variables declared in the environment (/compose/compose-file/#environment) section *override* these values – this holds true even if those values are empty or undefined.

```
env_file: .env
```

```
env_file:
  - ./common.env
  - ./apps/web.env
  - /opt/secrets.env
```

Compose expects each line in an env file to be in `VAR=VAL` format. Lines beginning with `#` are treated as comments and are ignored. Blank lines are also ignored.

```
# Set Rails/Rack environment
RACK_ENV=development
```

> **Note**: If your service specifies a build (/compose/compose-file/#build) option, variables defined in environment files are *not* automatically visible during the build. Use the args (/compose/compose-file/#args) sub-option of `build` to define build-time environment variables.

The value of `VAL` is used as is and not modified at all. For example if the value is surrounded by quotes (as is often the case of shell variables), the quotes are included in the value passed to Compose.

Keep in mind that *the order of files in the list is significant in determining the value assigned to a variable that shows up more than once*. The files in the list are processed from the top down. For the same variable specified in file `a.env` and assigned a different value in file `b.env`, if `b.env` is listed below (after), then the value from `b.env` stands. For example, given the following declaration in `docker-compose.yml`:

```
services:
  some-service:
    env_file:
      - a.env
      - b.env
```

And the following files:

```
# a.env
VAR=1
```

and

```
# b.env
VAR=hello
```

`$VAR` is `hello` .

environment

Add environment variables. You can use either an array or a dictionary. Any boolean values; true, false, yes no, need to be enclosed in quotes to ensure they are not converted to True or False by the YML parser.

Environment variables with only a key are resolved to their values on the machine Compose is running on, which can be helpful for secret or host-specific values.

```
environment:
  RACK_ENV: development
  SHOW: 'true'
  SESSION_SECRET:
```

```
environment:
  - RACK_ENV=development
  - SHOW=true
  - SESSION_SECRET
```

> **Note**: If your service specifies a build (/compose/compose-file/#build) option, variables defined in `environment` are *not* automatically visible during the build. Use the args (/compose/compose-file/#args) sub-option of `build` to define build-time environment variables.

expose

Expose ports without publishing them to the host machine - they'll only be accessible to linked services. Only the internal port can be specified.

```
expose:
 - "3000"
 - "8000"
```

external_links

Link to containers started outside this `docker-compose.yml` or even outside of Compose, especially for containers that provide shared or common services. `external_links` follow semantics similar to the legacy option `links` when specifying both the container name and the link alias (`CONTAINER:ALIAS`).

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

> **Notes:**
>
> If you're using the version 2 or above file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-2), the externally-created containers must be connected to at least one of the same networks as the service that is linking to them. Links (https://docs.docker.com/compose/compose-file/compose-file-v2#links) are a legacy option. We recommend using networks (/compose/compose-file/#networks) instead.
>
> This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

extra_hosts

Add hostname mappings. Use the same values as the docker client `--add-host` parameter.

```
extra_hosts:
 - "somehost:162.242.195.82"
 - "otherhost:50.31.209.229"
```

An entry with the ip address and hostname is created in `/etc/hosts` inside containers for this service, e.g:

```
162.242.195.82  somehost
50.31.209.229   otherhost
```

healthcheck

> Version 2.1 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-21) and up.

Configure a check that's run to determine whether or not containers for this service are "healthy". See the docs for the HEALTHCHECK Dockerfile instruction (https://docs.docker.com/engine/reference/builder/#healthcheck) for details on how healthchecks work.

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost"]
  interval: 1m30s
  timeout: 10s
  retries: 3
  start_period: 40s
```

`interval`, `timeout` and `start_period` are specified as durations (/compose/compose-file/#specifying-durations).

> **Note**: `start_period` is only supported for v3.4 and higher of the compose file format.

`test` must be either a string or a list. If it's a list, the first item must be either `NONE`, `CMD` or `CMD-SHELL`. If it's a string, it's equivalent to specifying `CMD-SHELL` followed by that string.

```
# Hit the local web app
test: ["CMD", "curl", "-f", "http://localhost"]
```

As above, but wrapped in `/bin/sh`. Both forms below are equivalent.

```
test: ["CMD-SHELL", "curl -f http://localhost || exit 1"]
```

```
test: curl -f https://localhost || exit 1
```

To disable any default healthcheck set by the image, you can use `disable: true`. This is equivalent to specifying `test: ["NONE"]`.

```
healthcheck:
  disable: true
```

image

Specify the image to start the container from. Can either be a repository/tag or a partial image ID.

```
image: redis
image: ubuntu:14.04
image: tutum/influxdb
image: example-registry.com:4000/postgresql
image: a4bc65fd
```

If the image does not exist, Compose attempts to pull it, unless you have also specified build (/compose/compose-file/#build), in which case it builds it using the specified options and tags it with the specified tag.

init

> Added in version 3.7 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-37).

Run an init inside the container that forwards signals and reaps processes. Set this option to `true` to enable this feature for the service.

```
version: "3.7"
services:
  web:
    image: alpine:latest
    init: true
```

> The default init binary that is used is Tini (https://github.com/krallin/tini), and is installed in `/usr/libexec/docker-init` on the daemon host. You can configure the daemon to use a custom init binary through the `init-path` configuration option (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file).

isolation

Specify a container's isolation technology. On Linux, the only supported value is `default`. On Windows, acceptable values are `default`, `process` and `hyperv`. Refer to the Docker Engine docs (https://docs.docker.com/engine/reference/commandline/run/#specify-isolation-technology-for-container---isolation) for details.

labels

Add metadata to containers using Docker labels (https://docs.docker.com/engine/userguide/labels-custom-metadata/). You can use either an array or a dictionary.

It's recommended that you use reverse-DNS notation to prevent your labels from conflicting with those used by other software.

```
labels:
  com.example.description: "Accounting webapp"
  com.example.department: "Finance"
  com.example.label-with-empty-value: ""
```

```
labels:
  - "com.example.description=Accounting webapp"
  - "com.example.department=Finance"
  - "com.example.label-with-empty-value"
```

links

> **Warning**: The `--link` flag is a legacy feature of Docker. It may eventually be removed. Unless you absolutely need to continue using it, we recommend that you use user-defined networks (https://docs.docker.com/engine/userguide/networking//#user-defined-networks) to facilitate communication between two containers instead of using `--link` . One feature that user-defined networks do not support that you can do with `--link` is sharing environmental variables between containers. However, you can use other mechanisms such as volumes to share environment variables between containers in a more controlled way.

Link to containers in another service. Either specify both the service name and a link alias (`SERVICE:ALIAS`), or just the service name.

```
web:
  links:
   - db
   - db:database
   - redis
```

Containers for the linked service are reachable at a hostname identical to the alias, or the service name if no alias was specified.

Links are not required to enable services to communicate - by default, any service can reach any other service at that service's name. (See also, the Links topic in Networking in Compose (https://docs.docker.com/compose/networking/#links).)

Links also express dependency between services in the same way as depends_on (/compose/compose-file/#depends_on), so they determine the order of service startup.

> **Notes**
>
> - If you define both links and networks (/compose/compose-file/#networks), services with links between them must share at least one network in common to communicate.
> - This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

logging

Logging configuration for the service.

```
logging:
  driver: syslog
  options:
    syslog-address: "tcp://192.168.0.42:123"
```

The `driver` name specifies a logging driver for the service's containers, as with the `--log-driver` option for docker run (documented here (https://docs.docker.com/engine/admin/logging/overview/)).

The default value is json-file.

```
driver: "json-file"
driver: "syslog"
driver: "none"
```

> **Note**: Only the `json-file` and `journald` drivers make the logs available directly from `docker-compose up` and `docker-compose logs`. Using any other driver does not print any logs.

Specify logging options for the logging driver with the `options` key, as with the `--log-opt` option for `docker run`.

Logging options are key-value pairs. An example of `syslog` options:

```
driver: "syslog"
options:
  syslog-address: "tcp://192.168.0.42:123"
```

The default driver json-file (https://docs.docker.com/engine/admin/logging/overview/#json-file), has options to limit the amount of logs stored. To do this, use a key-value pair for maximum storage size and maximum number of files:

```
options:
  max-size: "200k"
  max-file: "10"
```

The example shown above would store log files until they reach a `max-size` of 200kB, and then rotate them. The amount of individual log files stored is specified by the `max-file` value. As logs grow beyond the max limits, older log files are removed to allow storage of new logs.

Here is an example `docker-compose.yml` file that limits logging storage:

```
version: "3.7"
services:
  some-service:
    image: some-service
    logging:
      driver: "json-file"
      options:
        max-size: "200k"
        max-file: "10"
```

> **Logging options available depend on which logging driver you use**
>
> The above example for controlling log files and sizes uses options specific to the json-file driver (https://docs.docker.com/engine/admin/logging/overview/#json-file). These particular options are not available on other logging drivers. For a full list of supported logging drivers and their options, see logging drivers (https://docs.docker.com/engine/admin/logging/overview/).

network_mode

Network mode. Use the same values as the docker client `--network` parameter, plus the special form `service:[service name]` .

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"
```

> **Notes**
>
> - This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.
> - `network_mode: "host"` cannot be mixed with links (/compose/compose-file/#links).

networks

Networks to join, referencing entries under the top-level `networks` key (/compose/compose-file/#network-configuration-reference).

```
services:
  some-service:
    networks:
     - some-network
     - other-network
```

ALIASES

Aliases (alternative hostnames) for this service on the network. Other containers on the same network can use either the service name or this alias to connect to one of the service's containers.

Since `aliases` is network-scoped, the same service can have different aliases on different networks.

> **Note**: A network-wide alias can be shared by multiple containers, and even by multiple services. If it is, then exactly which container the name resolves to is not guaranteed.

The general format is shown here.

```
services:
  some-service:
    networks:
      some-network:
        aliases:
         - alias1
         - alias3
      other-network:
        aliases:
         - alias2
```

In the example below, three services are provided (`web` , `worker` , and `db`), along with two networks (`new` and `legacy`). The `db` service is reachable at the hostname `db` or `database` on the `new` network, and at `db` or `mysql` on the `legacy` network.

```
version: "3.7"

services:
  web:
    image: "nginx:alpine"
    networks:
      - new

  worker:
    image: "my-worker-image:latest"
    networks:
      - legacy

  db:
    image: mysql
    networks:
      new:
        aliases:
          - database
      legacy:
        aliases:
          - mysql

networks:
  new:
  legacy:
```

IPV4_ADDRESS, IPV6_ADDRESS

Specify a static IP address for containers for this service when joining the network.

The corresponding network configuration in the top-level networks section (/compose/compose-file/#network-configuration-reference) must have an `ipam` block with subnet configurations covering each static address.

> If IPv6 addressing is desired, the `enable_ipv6` (https://docs.docker.com/compose/compose-file/compose-file-v2/##enable_ipv6) option must be set, and you must use a version 2.x Compose file (https://docs.docker.com/compose/compose-file/compose-file-v2/#ipv4_address-ipv6_address). *IPv6 options do not currently work in swarm mode*.

An example:

```
version: "3.7"

services:
  app:
    image: nginx:alpine
    networks:
      app_net:
        ipv4_address: 172.16.238.10
        ipv6_address: 2001:3984:3989::10

networks:
  app_net:
    ipam:
      driver: default
      config:
        - subnet: "172.16.238.0/24"
        - subnet: "2001:3984:3989::/64"
```

pid

```
pid: "host"
```

Sets the PID mode to the host PID mode. This turns on sharing between container and the host operating system the PID address space. Containers launched with this flag can access and manipulate other containers in the bare-metal machine's namespace and vice versa.

ports

Expose ports.

> **Note**: Port mapping is incompatible with `network_mode: host`

SHORT SYNTAX

Either specify both ports (`HOST:CONTAINER`), or just the container port (an ephemeral host port is chosen).

> **Note**: When mapping ports in the `HOST:CONTAINER` format, you may experience erroneous results when using a container port lower than 60, because YAML parses numbers in the format `xx:yy` as a base-60 value. For this reason, we recommend always explicitly specifying your port mappings as strings.

```
ports:
 - "3000"
 - "3000-3005"
 - "8000:8000"
 - "9090-9091:8080-8081"
 - "49100:22"
 - "127.0.0.1:8001:8001"
 - "127.0.0.1:5000-5010:5000-5010"
 - "6060:6060/udp"
```

LONG SYNTAX

The long form syntax allows the configuration of additional fields that can't be expressed in the short form.

- `target` : the port inside the container
- `published` : the publicly exposed port
- `protocol` : the port protocol (`tcp` or `udp`)
- `mode` : `host` for publishing a host port on each node, or `ingress` for a swarm mode port to be load balanced.

```
ports:
  - target: 80
    published: 8080
    protocol: tcp
    mode: host
```

> **Note**: The long syntax is new in v3.2

restart

`no` is the default restart policy, and it does not restart a container under any circumstance. When `always` is specified, the container always restarts. The `on-failure` policy restarts a container if the exit code indicates an on-failure error.

```
restart: "no"
restart: always
restart: on-failure
restart: unless-stopped
```

> **Note**: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file. Use restart_policy (/compose/compose-file/#restart_policy) instead.

secrets

Grant access to secrets on a per-service basis using the per-service `secrets` configuration. Two different syntax variants are supported.

> **Note**: The secret must already exist or be defined in the top-level `secrets` configuration (/compose/compose-file/#secrets-configuration-reference) of this stack file, or stack deployment fails.

For more information on secrets, see secrets (https://docs.docker.com/engine/swarm/secrets/).

SHORT SYNTAX

The short syntax variant only specifies the secret name. This grants the container access to the secret and mounts it at `/run/secrets/<secret_name>` within the container. The source name and destination mountpoint are both set to the secret name.

The following example uses the short syntax to grant the `redis` service access to the `my_secret` and `my_other_secret` secrets. The value of `my_secret` is set to the contents of the file `./my_secret.txt`, and `my_other_secret` is defined as an external resource, which means that it has already been defined in Docker, either by running the `docker secret create` command or by another stack deployment. If the external secret does not exist, the stack deployment fails with a `secret not found` error.

```
version: "3.7"
services:
  redis:
    image: redis:latest
    deploy:
      replicas: 1
    secrets:
      - my_secret
      - my_other_secret
secrets:
  my_secret:
    file: ./my_secret.txt
  my_other_secret:
    external: true
```

LONG SYNTAX

The long syntax provides more granularity in how the secret is created within the service's task containers.

- `source` : The name of the secret as it exists in Docker.
- `target` : The name of the file to be mounted in `/run/secrets/` in the service's task containers. Defaults to `source` if not specified.
- `uid` and `gid` : The numeric UID or GID that owns the file within `/run/secrets/` in the service's task containers. Both default to `0` if not specified.
- `mode` : The permissions for the file to be mounted in `/run/secrets/` in the service's task containers, in octal notation. For instance, `0444` represents world-readable. The default in Docker 1.13.1 is `0000` , but is be `0444` in newer versions. Secrets cannot be writable because they are mounted in a temporary filesystem, so if you set the writable bit, it is ignored. The executable bit can be set. If you aren't familiar with UNIX file permission modes, you may find this permissions calculator (http://permissions-calculator.org/) useful.

The following example sets name of the `my_secret` to `redis_secret` within the container, sets the mode to `0440` (group-readable) and sets the user and group to `103` . The `redis` service does not have access to the `my_other_secret` secret.

```
version: "3.7"
services:
  redis:
    image: redis:latest
    deploy:
      replicas: 1
    secrets:
      - source: my_secret
        target: redis_secret
        uid: '103'
        gid: '103'
        mode: 0440
secrets:
  my_secret:
    file: ./my_secret.txt
  my_other_secret:
    external: true
```

You can grant a service access to multiple secrets and you can mix long and short syntax. Defining a secret does not imply granting a service access to it.

security_opt

Override the default labeling scheme for each container.

```
security_opt:
  - label:user:USER
  - label:role:ROLE
```

> Note: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

stop_grace_period

Specify how long to wait when attempting to stop a container if it doesn't handle SIGTERM (or whatever stop signal has been specified with `stop_signal` (/compose/compose-file/#stopsignal)), before sending SIGKILL. Specified as a duration (/compose/compose-file/#specifying-durations).

```
stop_grace_period: 1s
stop_grace_period: 1m30s
```

By default, `stop` waits 10 seconds for the container to exit before sending SIGKILL.

stop_signal

Sets an alternative signal to stop the container. By default `stop` uses SIGTERM. Setting an alternative signal using `stop_signal` causes `stop` to send that signal instead.

```
stop_signal: SIGUSR1
```

sysctls

Kernel parameters to set in the container. You can use either an array or a dictionary.

```
sysctls:
  net.core.somaxconn: 1024
  net.ipv4.tcp_syncookies: 0
```

```
sysctls:
  - net.core.somaxconn=1024
  - net.ipv4.tcp_syncookies=0
```

> **Note**: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

tmpfs

> Version 2 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-2) and up.

Mount a temporary file system inside the container. Can be a single value or a list.

```
tmpfs: /run
```

```
tmpfs:
  - /run
  - /tmp
```

> **Note**: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3-3.5) Compose file.

> Version 3.6 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-3) and up.

Mount a temporary file system inside the container. Size parameter specifies the size of the tmpfs mount in bytes. Unlimited by default.

```
 - type: tmpfs
     target: /app
     tmpfs:
       size: 1000
```

ulimits

Override the default ulimits for a container. You can either specify a single limit as an integer or soft/hard limits as a mapping.

```
ulimits:
  nproc: 65535
  nofile:
    soft: 20000
    hard: 40000
```

userns_mode

```
userns_mode: "host"
```

Disables the user namespace for this service, if Docker daemon is configured with user namespaces. See dockerd (https://docs.docker.com/engine/reference/commandline/dockerd/#disable-user-

namespace-for-a-container) for more information.

> **Note**: This option is ignored when deploying a stack in swarm mode (https://docs.docker.com/engine/reference/commandline/stack_deploy/) with a (version 3) Compose file.

volumes

Mount host paths or named volumes, specified as sub-options to a service.

You can mount a host path as part of a definition for a single service, and there is no need to define it in the top level `volumes` key.

But, if you want to reuse a volume across multiple services, then define a named volume in the top-level `volumes` key (/compose/compose-file/#volume-configuration-reference). Use named volumes with services, swarms, and stack files (/compose/compose-file/#volumes-for-services-swarms-and-stack-files).

> **Note**: The top-level volumes (/compose/compose-file/#volume-configuration-reference) key defines a named volume and references it from each service's `volumes` list. This replaces `volumes_from` in earlier versions of the Compose file format. See Use volumes (https://docs.docker.com/engine/admin/volumes/volumes/) and Volume Plugins (https://docs.docker.com/engine/extend/plugins_volume/) for general information on volumes.

This example shows a named volume (`mydata`) being used by the `web` service, and a bind mount defined for a single service (first path under `db` service `volumes`). The `db` service also uses a named volume called `dbdata` (second path under `db` service `volumes`), but defines it using the old string format for mounting a named volume. Named volumes must be listed under the top-level `volumes` key, as shown.

```
version: "3.7"
services:
  web:
    image: nginx:alpine
    volumes:
      - type: volume
        source: mydata
        target: /data
        volume:
          nocopy: true
      - type: bind
        source: ./static
        target: /opt/app/static

  db:
    image: postgres:latest
    volumes:
      - "/var/run/postgres/postgres.sock:/var/run/postgres/postgres.sock"
      - "dbdata:/var/lib/postgresql/data"

volumes:
  mydata:
  dbdata:
```

> **Note**: See Use volumes (https://docs.docker.com/engine/admin/volumes/volumes/) and Volume Plugins (https://docs.docker.com/engine/extend/plugins_volume/) for general information on volumes.

SHORT SYNTAX

Optionally specify a path on the host machine (`HOST:CONTAINER`), or an access mode (`HOST:CONTAINER:ro`).

You can mount a relative path on the host, that expands relative to the directory of the Compose configuration file being used. Relative paths should always begin with `.` or `..` .

```
volumes:
  # Just specify a path and let the Engine create a volume
  - /var/lib/mysql

  # Specify an absolute path mapping
  - /opt/data:/var/lib/mysql

  # Path on the host, relative to the Compose file
  - ./cache:/tmp/cache

  # User-relative path
  - ~/configs:/etc/configs/:ro

  # Named volume
  - datavolume:/var/lib/mysql
```

LONG SYNTAX

The long form syntax allows the configuration of additional fields that can't be expressed in the short form.

- `type` : the mount type `volume` , `bind` or `tmpfs`
- `source` : the source of the mount, a path on the host for a bind mount, or the name of a volume defined in the top-level `volumes` key (/compose/compose-file/#volume-configuration-reference). Not applicable for a tmpfs mount.
- `target` : the path in the container where the volume is mounted
- `read_only` : flag to set the volume as read-only
- `bind` : configure additional bind options
  - `propagation` : the propagation mode used for the bind
- `volume` : configure additional volume options
  - `nocopy` : flag to disable copying of data from a container when a volume is created
- `tmpfs` : configure additional tmpfs options
  - `size` : the size for the tmpfs mount in bytes
- `consistency` : the consistency requirements of the mount, one of `consistent` (host and container have identical view), `cached` (read cache, host view is authoritative) or `delegated` (read-write cache, container's view is authoritative)

```
version: "3.7"
services:
  web:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - type: volume
        source: mydata
        target: /data
        volume:
          nocopy: true
      - type: bind
        source: ./static
        target: /opt/app/static

networks:
  webnet:

volumes:
  mydata:
```

> **Note**: The long syntax is new in v3.2

VOLUMES FOR SERVICES, SWARMS, AND STACK FILES

When working with services, swarms, and `docker-stack.yml` files, keep in mind that the tasks (containers) backing a service can be deployed on any node in a swarm, and this may be a different node each time the service is updated.

In the absence of having named volumes with specified sources, Docker creates an anonymous volume for each task backing a service. Anonymous volumes do not persist after the associated containers are removed.

If you want your data to persist, use a named volume and a volume driver that is multi-host aware, so that the data is accessible from any node. Or, set constraints on the service so that its tasks are deployed on a node that has the volume present.

As an example, the `docker-stack.yml` file for the votingapp sample in Docker Labs (https://github.com/docker/labs/blob/master/beginner/chapters/votingapp.md) defines a service called `db` that runs a `postgres` database. It is configured as a named volume to persist the data on the swarm, *and* is constrained to run only on `manager` nodes. Here is the relevant snip-it from that file:

```
version: "3.7"
services:
  db:
    image: postgres:9.4
    volumes:
      - db-data:/var/lib/postgresql/data
    networks:
      - backend
    deploy:
      placement:
        constraints: [node.role == manager]
```

CACHING OPTIONS FOR VOLUME MOUNTS (DOCKER DESKTOP FOR MAC)

On Docker 17.04 CE Edge and up, including 17.06 CE Edge and Stable, you can configure container-and-host consistency requirements for bind-mounted directories in Compose files to allow for better performance on read/write of volume mounts. These options address issues specific to `osxfs` file sharing, and therefore are only applicable on Docker Desktop for Mac.

The flags are:

- `consistent` : Full consistency. The container runtime and the host maintain an identical view of the mount at all times. This is the default.
- `cached` : The host's view of the mount is authoritative. There may be delays before updates made on the host are visible within a container.
- `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.

Here is an example of configuring a volume as `cached` :

```
version: "3.7"
services:
  php:
    image: php:7.1-fpm
    ports:
      - "9000"
    volumes:
      - .:/var/www/project:cached
```

Full detail on these flags, the problems they solve, and their `docker run` counterparts is in the Docker Desktop for Mac topic Performance tuning for volume mounts (shared filesystems) (https://docs.docker.com/docker-for-mac/osxfs-caching/).

domainname, hostname, ipc, mac_address, privileged, read_only, shm_size, stdin_open, tty, user, working_dir

Each of these is a single value, analogous to its docker run (https://docs.docker.com/engine/reference/run/) counterpart. Note that `mac_address` is a legacy option.

```
user: postgresql
working_dir: /code

domainname: foo.com
hostname: foo
ipc: host
mac_address: 02:42:ac:11:65:43

privileged: true


read_only: true
shm_size: 64M
stdin_open: true
tty: true
```

Specifying durations

Some configuration options, such as the `interval` and `timeout` sub-options for `check` (/compose/compose-file/#healthcheck), accept a duration as a string in a format that looks like this:

```
2.5s
10s
1m30s
2h32m
5h34m56s
```

The supported units are `us`, `ms`, `s`, `m` and `h`.

Specifying byte values

Some configuration options, such as the `shm_size` sub-option for `build` (/compose/compose-file/#build), accept a byte value as a string in a format that looks like this:

```
2b
1024kb
2048k
300m
1gb
```

The supported units are `b`, `k`, `m` and `g`, and their alternative notation `kb`, `mb` and `gb`. Decimal values are not supported at this time.

Volume configuration reference

While it is possible to declare volumes (/compose/compose-file/#volumes) on the file as part of the service declaration, this section allows you to create named volumes (without relying on `volumes_from`) that can be reused across multiple services, and are easily retrieved and inspected using the docker command line or API. See the docker volume (https://docs.docker.com/engine/reference/commandline/volume_create/) subcommand documentation for more information.

See Use volumes (https://docs.docker.com/engine/admin/volumes/volumes/) and Volume Plugins (https://docs.docker.com/engine/extend/plugins_volume/) for general information on volumes.

Here's an example of a two-service setup where a database's data directory is shared with another service as a volume so that it can be periodically backed up:

```
version: "3.7"

services:
  db:
    image: db
    volumes:
      - data-volume:/var/lib/db
  backup:
    image: backup-service
    volumes:
      - data-volume:/var/lib/backup/data

volumes:
  data-volume:
```

An entry under the top-level `volumes` key can be empty, in which case it uses the default driver configured by the Engine (in most cases, this is the `local` driver). Optionally, you can configure it with the following keys:

driver

Specify which volume driver should be used for this volume. Defaults to whatever driver the Docker Engine has been configured to use, which in most cases is `local`. If the driver is not available, the Engine returns an error when `docker-compose up` tries to create the volume.

```
driver: foobar
```

driver_opts

Specify a list of options as key-value pairs to pass to the driver for this volume. Those options are driver-dependent - consult the driver's documentation for more information. Optional.

```
volumes:
  example:
    driver_opts:
      type: "nfs"
      o: "addr=10.40.0.199,nolock,soft,rw"
      device: ":/docker/example"
```

external

If set to `true`, specifies that this volume has been created outside of Compose. `docker-compose up` does not attempt to create it, and raises an error if it doesn't exist.

For version 3.3 and below of the format, `external` cannot be used in conjunction with other volume configuration keys (`driver`, `driver_opts`, `labels`). This limitation no longer exists for version 3.4 (https://docs.docker.com/compose/compose-file/compose-versioning/#version-34) and above.

In the example below, instead of attempting to create a volume called `[projectname]_data`, Compose looks for an existing volume simply called `data` and mount it into the `db` service's containers.

```
version: "3.7"

services:
  db:
    image: postgres
    volumes:
      - data:/var/lib/postgresql/data

volumes:
  data:
    external: true
```

> external.name was deprecated in version 3.4 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-34) use `name` instead.

You can also specify the name of the volume separately from the name used to refer to it within the Compose file:

```
volumes:
  data:
    external:
      name: actual-name-of-volume
```

> **External volumes are always created with docker stack deploy**
>
> External volumes that do not exist *are created* if you use docker stack deploy (/compose/compose-file/#deploy) to launch the app in swarm mode (https://docs.docker.com/engine/swarm/) (instead of docker compose up (https://docs.docker.com/compose/reference/up/)). In swarm mode, a volume is automatically created when it is defined by a service. As service tasks are scheduled on new nodes, swarmkit (https://github.com/docker/swarmkit/blob/master/README.md) creates the volume on the local node. To learn more, see moby/moby#29976 (https://github.com/moby/moby/issues/29976).

labels

Add metadata to containers using Docker labels (https://docs.docker.com/engine/userguide/labels-custom-metadata/). You can use either an array or a dictionary.

It's recommended that you use reverse-DNS notation to prevent your labels from conflicting with those used by other software.

```
labels:
  com.example.description: "Database volume"
  com.example.department: "IT/Ops"
  com.example.label-with-empty-value: ""
```

```
labels:
  - "com.example.description=Database volume"
  - "com.example.department=IT/Ops"
  - "com.example.label-with-empty-value"
```

name

> Added in version 3.4 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-34)

Set a custom name for this volume. The name field can be used to reference volumes that contain special characters. The name is used as is and will **not** be scoped with the stack name.

```
version: "3.7"
volumes:
  data:
    name: my-app-data
```

It can also be used in conjunction with the `external` property:

```
version: "3.7"
volumes:
  data:
    external: true
    name: my-app-data
```

Network configuration reference

The top-level `networks` key lets you specify networks to be created.

- For a full explanation of Compose's use of Docker networking features and all network driver options, see the Networking guide (https://docs.docker.com/compose/networking/).
- For Docker Labs (https://github.com/docker/labs/blob/master/README.md) tutorials on networking, start with Designing Scalable, Portable Docker Container Networks (https://github.com/docker/labs/blob/master/networking/README.md)

driver

Specify which driver should be used for this network.

The default driver depends on how the Docker Engine you're using is configured, but in most instances it is `bridge` on a single host and `overlay` on a Swarm.

The Docker Engine returns an error if the driver is not available.

```
driver: overlay
```

BRIDGE

Docker defaults to using a `bridge` network on a single host. For examples of how to work with bridge networks, see the Docker Labs tutorial on Bridge networking (https://github.com/docker/labs/blob/master/networking/A2-bridge-networking.md).

OVERLAY

The `overlay` driver creates a named network across multiple nodes in a swarm (https://docs.docker.com/engine/swarm/).

- For a working example of how to build and use an `overlay` network with a service in swarm mode, see the Docker Labs tutorial on Overlay networking and service discovery (https://github.com/docker/labs/blob/master/networking/A3-overlay-networking.md).
- For an in-depth look at how it works under the hood, see the networking concepts lab on the Overlay Driver Network Architecture (https://github.com/docker/labs/blob/master/networking/concepts/06-overlay-networks.md).

HOST OR NONE

Use the host's networking stack, or no networking. Equivalent to `docker run --net=host` or `docker run --net=none`. Only used if you use `docker stack` commands. If you use the `docker-compose` command, use network_mode (/compose/compose-file/#network_mode) instead.

If you want to use a particular network on a common build, use [network] as mentioned in the second yaml file example.

The syntax for using built-in networks such as `host` and `none` is a little different. Define an external network with the name `host` or `none` (that Docker has already created automatically) and an alias that Compose can use (`hostnet` or `nonet` in the following examples), then grant the service access to that network using the alias.

```
version: "3.7"
services:
  web:
    networks:
      hostnet: {}

networks:
  hostnet:
    external: true
    name: host
```

```
services:
  web:
    ...
    build:
      ...
      network: host
      context: .
      ...
```

```
services:
  web:
    ...
    networks:
      nonet: {}

networks:
  nonet:
    external: true
    name: none
```

driver_opts

Specify a list of options as key-value pairs to pass to the driver for this network. Those options are driver-dependent - consult the driver's documentation for more information. Optional.

```
driver_opts:
  foo: "bar"
  baz: 1
```

attachable

> **Note**: Only supported for v3.2 and higher.

Only used when the `driver` is set to `overlay`. If set to `true`, then standalone containers can attach to this network, in addition to services. If a standalone container attaches to an overlay network, it can communicate with services and standalone containers that are also attached to the overlay network from other Docker daemons.

```
networks:
  mynet1:
    driver: overlay
    attachable: true
```

enable_ipv6

Enable IPv6 networking on this network.

> ✖ Not supported in Compose File version 3
>
> `enable_ipv6` requires you to use a version 2 Compose file, as this directive is not yet supported in Swarm mode.

ipam

Specify custom IPAM config. This is an object with several properties, each of which is optional:

- `driver` : Custom IPAM driver, instead of the default.
- `config` : A list with zero or more config blocks, each containing any of the following keys:
  - `subnet` : Subnet in CIDR format that represents a network segment

A full example:

```
ipam:
  driver: default
  config:
    - subnet: 172.28.0.0/16
```

> **Note**: Additional IPAM configurations, such as `gateway` , are only honored for version 2 at the moment.

internal

By default, Docker also connects a bridge network to it to provide external connectivity. If you want to create an externally isolated overlay network, you can set this option to `true` .

labels

Add metadata to containers using Docker labels (https://docs.docker.com/engine/userguide/labels-custom-metadata/). You can use either an array or a dictionary.

It's recommended that you use reverse-DNS notation to prevent your labels from conflicting with those used by other software.

```
labels:
  com.example.description: "Financial transaction network"
  com.example.department: "Finance"
  com.example.label-with-empty-value: ""
```

```
labels:
  - "com.example.description=Financial transaction network"
  - "com.example.department=Finance"
  - "com.example.label-with-empty-value"
```

external

If set to `true`, specifies that this network has been created outside of Compose. `docker-compose up` does not attempt to create it, and raises an error if it doesn't exist.

For version 3.3 and below of the format, `external` cannot be used in conjunction with other network configuration keys (`driver` , `driver_opts` , `ipam` , `internal`). This limitation no longer exists for version 3.4 (https://docs.docker.com/compose/compose-file/compose-versioning/#version-34) and above.

In the example below, `proxy` is the gateway to the outside world. Instead of attempting to create a network called `[projectname]_outside` , Compose looks for an existing network simply called `outside` and connect the `proxy` service's containers to it.

```
version: "3.7"

services:
  proxy:
    build: ./proxy
    networks:
      - outside
      - default
  app:
    build: ./app
    networks:
      - default

networks:
  outside:
    external: true
```

> external.name was deprecated in version 3.5 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-35) use `name` instead.

You can also specify the name of the network separately from the name used to refer to it within the Compose file:

```
version: "3.7"
networks:
  outside:
    external:
      name: actual-name-of-network
```

name

> Added in version 3.5 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-35)

Set a custom name for this network. The name field can be used to reference networks which contain special characters. The name is used as is and will **not** be scoped with the stack name.

```
version: "3.7"
networks:
  network1:
    name: my-app-net
```

It can also be used in conjunction with the `external` property:

```
version: "3.7"
networks:
  network1:
    external: true
    name: my-app-net
```

configs configuration reference

The top-level `configs` declaration defines or references configs (https://docs.docker.com/engine/swarm/configs/) that can be granted to the services in this stack. The source of the config is either `file` or `external`.

- `file`: The config is created with the contents of the file at the specified path.
- `external`: If set to true, specifies that this config has already been created. Docker does not attempt to create it, and if it does not exist, a `config not found` error occurs.
- `name`: The name of the config object in Docker. This field can be used to reference configs that contain special characters. The name is used as is and will **not** be scoped with the stack name. Introduced in version 3.5 file format.

In this example, `my_first_config` is created (as `<stack_name>_my_first_config)` when the stack is deployed, and `my_second_config` already exists in Docker.

```
configs:
  my_first_config:
    file: ./config_data
  my_second_config:
    external: true
```

Another variant for external configs is when the name of the config in Docker is different from the name that exists within the service. The following example modifies the previous one to use the external config called `redis_config`.

```
configs:
  my_first_config:
    file: ./config_data
  my_second_config:
    external:
      name: redis_config
```

You still need to grant access to the config (/compose/compose-file/#configs) to each service in the stack.

secrets configuration reference

The top-level `secrets` declaration defines or references secrets (https://docs.docker.com/engine/swarm/secrets/) that can be granted to the services in this stack. The source of the secret is either `file` or `external`.

- `file`: The secret is created with the contents of the file at the specified path.

- `external` : If set to true, specifies that this secret has already been created. Docker does not attempt to create it, and if it does not exist, a `secret not found` error occurs.
- `name` : The name of the secret object in Docker. This field can be used to reference secrets that contain special characters. The name is used as is and will **not** be scoped with the stack name. Introduced in version 3.5 file format.

In this example, `my_first_secret` is created as `<stack_name>_my_first_secret` when the stack is deployed, and `my_second_secret` already exists in Docker.

```
secrets:
  my_first_secret:
    file: ./secret_data
  my_second_secret:
    external: true
```

Another variant for external secrets is when the name of the secret in Docker is different from the name that exists within the service. The following example modifies the previous one to use the external secret called `redis_secret` .

Compose File v3.5 and above

```
secrets:
  my_first_secret:
    file: ./secret_data
  my_second_secret:
    external: true
    name: redis_secret
```

Compose File v3.4 and under

```
  my_second_secret:
    external:
      name: redis_secret
```

You still need to grant access to the secrets (/compose/compose-file/#secrets) to each service in the stack.

Variable substitution

Your configuration options can contain environment variables. Compose uses the variable values from the shell environment in which `docker-compose` is run. For example, suppose the shell contains `POSTGRES_VERSION=9.3` and you supply this configuration:

```
db:
  image: "postgres:${POSTGRES_VERSION}"
```

When you run `docker-compose up` with this configuration, Compose looks for the `POSTGRES_VERSION` environment variable in the shell and substitutes its value in. For this example, Compose resolves the `image` to `postgres:9.3` before running the configuration.

If an environment variable is not set, Compose substitutes with an empty string. In the example above, if `POSTGRES_VERSION` is not set, the value for the `image` option is `postgres:` .

You can set default values for environment variables using a `.env` file (https://docs.docker.com/compose/env-file/), which Compose automatically looks for. Values set in the shell environment override those set in the `.env` file.

> **Important**: The `.env file` feature only works when you use the `docker-compose up` command and does not work with `docker stack deploy` .

Both `$VARIABLE` and `${VARIABLE}` syntax are supported. Additionally when using the 2.1 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-21), it is possible to provide inline default values using typical shell syntax:

- `${VARIABLE:-default}` evaluates to `default` if `VARIABLE` is unset or empty in the environment.
- `${VARIABLE-default}` evaluates to `default` only if `VARIABLE` is unset in the environment.

Similarly, the following syntax allows you to specify mandatory variables:

- `${VARIABLE:?err}` exits with an error message containing `err` if `VARIABLE` is unset or empty in the environment.
- `${VARIABLE?err}` exits with an error message containing `err` if `VARIABLE` is unset in the environment.

Other extended shell-style features, such as `${VARIABLE/foo/bar}` , are not supported.

You can use a `$$` (double-dollar sign) when your configuration needs a literal dollar sign. This also prevents Compose from interpolating a value, so a `$$` allows you to refer to environment variables that you don't want processed by Compose.

```
web:
  build: .
  command: "$$VAR_NOT_INTERPOLATED_BY_COMPOSE"
```

If you forget and use a single dollar sign (`$`), Compose interprets the value as an environment variable and warns you:

The VAR_NOT_INTERPOLATED_BY_COMPOSE is not set. Substituting an empty string.

Extension fields

> Added in version 3.4 file format (https://docs.docker.com/compose/compose-file/compose-versioning/#version-34).

It is possible to re-use configuration fragments using extension fields. Those special fields can be of any format as long as they are located at the root of your Compose file and their name start with the `x-` character sequence.

> **Note**
>
> Starting with the 3.7 format (for the 3.x series) and 2.4 format (for the 2.x series), extension fields are also allowed at the root of service, volume, network, config and secret definitions.

```
version: '3.4'
x-custom:
  items:
    - a
    - b
  options:
    max-size: '12m'
  name: "custom"
```

The contents of those fields are ignored by Compose, but they can be inserted in your resource definitions using YAML anchors (http://www.yaml.org/spec/1.2/spec.html#id2765878). For example, if you want several

of your services to use the same logging configuration:

```
logging:
  options:
    max-size: '12m'
    max-file: '5'
  driver: json-file
```

You may write your Compose file as follows:

```
version: '3.4'
x-logging:
  &default-logging
  options:
    max-size: '12m'
    max-file: '5'
  driver: json-file

services:
  web:
    image: myapp/web:latest
    logging: *default-logging
  db:
    image: mysql:latest
    logging: *default-logging
```

It is also possible to partially override values in extension fields using the YAML merge type (http://yaml.org/type/merge.html). For example:

```
version: '3.4'
x-volumes:
  &default-volume
  driver: foobar-storage

services:
  web:
    image: myapp/web:latest
    volumes: ["vol1", "vol2", "vol3"]
volumes:
  vol1: *default-volume
  vol2:
    << : *default-volume
    name: volume02
  vol3:
    << : *default-volume
    driver: default
    name: volume-local
```

Compose documentation

- User guide (https://docs.docker.com/compose/)
- Installing Compose (https://docs.docker.com/compose/install/)
- Compose file versions and upgrading (https://docs.docker.com/compose/compose-file/compose-versioning/)
- Get started with Docker (https://docs.docker.com/get-started/)
- Samples (https://docs.docker.com/samples/)
- Command line reference (https://docs.docker.com/compose/reference/)

fig (https://docs.docker.com/glossary/?term=fig), composition (https://docs.docker.com/glossary/?term=composition), compose (https://docs.docker.com/glossary/?term=compose), docker (https://docs.docker.com/glossary/?term=docker)