Services top-level elements

Table of contents

- annotations
- attach
- build
- blkio_config
  - device_read_bps, device_write_bps
  - device_read_iops, device_write_iops
  - weight
  - weight_device
- cpu_count
- cpu_percent
- cpu_shares
- cpu_period
- cpu_quota
- cpu_rt_runtime
- cpu_rt_period
- cpus
- cpuset
- cap_add
- cap_drop
- cgroup
- cgroup_parent
- command
- configs
  - Short syntax
  - Long syntax
- container_name
- credential_spec
  - Example gMSA configuration
- depends_on
  - Short syntax
  - Long syntax
- deploy
- develop
- device_cgroup_rules
- devices
- dns
- dns_opt
- dns_search
- domainname
- entrypoint
- env_file
  - Env_file format
- environment
- expose
- extends
  - Finding referenced service
  - Merging service definitions
- external_links
- extra_hosts
  - Short syntax
  - Long syntax
- group_add
- healthcheck
- hostname
- image
- init

- ipc
- isolation
- labels
- links
- logging
- mac_address
- mem_limit
- mem_reservation
- mem_swappiness
- memswap_limit
- network_mode
- networks
  - aliases
  - ipv4_address, ipv6_address
  - link_local_ips
  - mac_address
  - priority
- oom_kill_disable
- oom_score_adj
- pid
- pids_limit
- platform
- ports
  - Short syntax
  - Long syntax
- privileged
- profiles
- pull_policy
- read_only
- restart
- runtime
- scale
- secrets
  - Short syntax
  - Long syntax
- security_opt
- shm_size
- stdin_open
- stop_grace_period
- stop_signal
- storage_opt
- sysctls
- tmpfs
- tty
- ulimits
- user
- userns_mode
- uts
- volumes
  - Short syntax
  - Long syntax
- volumes_from
- working_dir

A service is an abstract definition of a computing resource within an application which can be scaled or replaced independently from other components. Services are backed by a set of containers, run by the platform according to replication requirements and placement constraints. As services are backed by containers, they are defined by a Docker image and set of runtime arguments. All containers within a service are identically created with these arguments.

A Compose file must declare a `services` top-level element as a map whose keys are string representations of service names, and whose values are service definitions. A service definition contains the configuration that is applied to each service container.

Each service may also include a `build` section, which defines how to create the Docker image for the service. Compose supports building docker images using this service definition. If not used, the `build` section is ignored and the Compose file is still considered valid. Build support is an optional aspect of the Compose Specification, and is described in detail in the Compose Build Specification documentation.

Each service defines runtime constraints and requirements to run its containers. The `deploy` section groups these constraints and allows the platform to adjust the deployment strategy to best match containers' needs with available resources. Deploy support is an optional aspect of the Compose Specification, and is described in detail in the Compose Deploy Specification documentation. If not implemented the `deploy` section is ignored and the Compose file is still considered valid.

annotations

`annotations` defines annotations for the container. `annotations` can use either an array or a map.

```
annotations:
 com.example.foo: bar

annotations:
 - com.example.foo=bar
```

attach

Introduced in Docker Compose version 2.20.0

When `attach` is defined and set to `false` Compose does not collect service logs, until you explicitly request it to.

The default service configuration is `attach: true`.

build

`build` specifies the build configuration for creating a container image from source, as defined in the Compose Build Specification.

blkio_config

`blkio_config` defines a set of configuration options to set block IO limits for a service.

```
services:
 foo:
   image: busybox
   blkio_config:
      weight: 300
      weight_device:
        - path: /dev/sda
          weight: 400
      device_read_bps:
        - path: /dev/sdb
          rate: '12mb'
      device_read_iops:
        - path: /dev/sdb
          rate: 120
      device_write_bps:
        - path: /dev/sdb
          rate: '1024k'
      device_write_iops:
        - path: /dev/sdb
          rate: 30
```

device_read_bps, device_write_bps

Set a limit in bytes per second for read / write operations on a given device. Each item in the list must have two keys:

- `path`: Defines the symbolic path to the affected device.
- `rate`: Either as an integer value representing the number of bytes or as a string expressing a byte value.

device_read_iops, device_write_iops

Set a limit in operations per second for read / write operations on a given device. Each item in the list must have two keys:

- `path`: Defines the symbolic path to the affected device.
- `rate`: As an integer value representing the permitted number of operations per second.

weight

Modify the proportion of bandwidth allocated to a service relative to other services. Takes an integer value between 10 and 1000, with 500 being the default.

weight_device

Fine-tune bandwidth allocation by device. Each item in the list must have two keys:

- `path`: Defines the symbolic path to the affected device.
- `weight`: An integer value between 10 and 1000.

cpu_count

`cpu_count` defines the number of usable CPUs for service container.

cpu_percent

`cpu_percent` defines the usable percentage of the available CPUs.

cpu_shares

`cpu_shares` defines, as integer value, a service container's relative CPU weight versus other containers.

cpu_period

`cpu_period` configures CPU CFS (Completely Fair Scheduler) period when a platform is based on Linux kernel.

cpu_quota

`cpu_quota` configures CPU CFS (Completely Fair Scheduler) quota when a platform is based on Linux kernel.

cpu_rt_runtime

`cpu_rt_runtime` configures CPU allocation parameters for platforms with support for realtime scheduler. It can be either an integer value using microseconds as unit or a duration.

```
 cpu_rt_runtime: '400ms'
cpu_rt_runtime: 95000`
```

cpu_rt_period

`cpu_rt_period` configures CPU allocation parameters for platforms with support for realtime scheduler. It can be either an integer value using microseconds as unit or a duration.

```
 cpu_rt_period: '1400us'
cpu_rt_period: 11000`
```

cpus

`cpus` define the number of (potentially virtual) CPUs to allocate to service containers. This is a fractional number. `0.000` means no limit.

When set, `cpus` must be consistent with the `cpus` attribute in the Deploy Specification.

cpuset

`cpuset` defines the explicit CPUs in which to allow execution. Can be a range `0-3` or a list `0,1`

cap_add

`cap_add` specifies additional container capabilities as strings.

```
cap_add:
 - ALL
```

cap_drop

`cap_drop` specifies container capabilities to drop as strings.

```
cap_drop:
 - NET_ADMIN
 - SYS_ADMIN
```

cgroup

Introduced in Docker Compose version 2.15.0

`cgroup` specifies the cgroup namespace to join. When unset, it is the container runtime's decision to select which cgroup namespace to use, if supported.

- `host`: Runs the container in the Container runtime cgroup namespace.
- `private`: Runs the container in its own private cgroup namespace.

cgroup_parent

`cgroup_parent` specifies an optional parent cgroup for the container.

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

command

`command` overrides the default command declared by the container image, for example by Dockerfile's `CMD`.

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

The value can also be a list, in a manner similar to Dockerfile:

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

If the value is `null`, the default command from the image is used.

If the value is `[]` (empty list) or `''` (empty string), the default command declared by the image is ignored, i.e. overridden to be empty.

configs

Configs allow services to adapt their behaviour without the need to rebuild a Docker image. Services can only access configs when explicitly granted by the `configs` attribute. Two different syntax variants are supported.

Compose reports an error if `config` doesn't exist on the platform or isn't defined in the `configs` top-level element in the Compose file.

There are two syntaxes defined for configs: a short syntax and a long syntax.

You can grant a service access to multiple configs, and you can mix long and short syntax.

Short syntax

The short syntax variant only specifies the config name. This grants the container access to the config and mounts it as files into a serviceâ■■s containerâ■■s filesystem. The location of the mount point within the container defaults to `/<config_name>` in Linux containers, and `C:\<config-name>` in Windows containers.

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 the platform. If the external config does not exist, the deployment fails.

```
services:
 redis:
   image: redis:latest
   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 the platform.
- `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 the service's task containers. Default value when not specified is USER running container.
- `mode`: The permissions for the file that is mounted within the service's task containers, in octal notation. Default value is world-readable (`0444`). Writable bit must be ignored. The executable bit can be set.

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.

```
services:
 redis:
   image: redis:latest
   configs:
     - source: my_config
       target: /redis_config
       uid: "103"
       gid: "103"
       mode: 0440
configs:
 my_config:
   external: true
 my_other_config:
   external: true
```

container_name

`container_name` is a string that specifies a custom container name, rather than a name generated by default.

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

Compose does not scale a service beyond one container if the Compose file specifies a `container_name`. Attempting to do so results in an error.

`container_name` follows the regex format of `[a-zA-Z0-9][a-zA-Z0-9_.-]+`

credential_spec

`credential_spec` configures the credential spec for a managed service account.

If you have services that use Windows containers, you can use `file:` and `registry:` protocols for `credential_spec`. Compose also supports additional protocols for custom use-cases.

The `credential_spec` must be in the format `file://<filename>` or `registry://<value-name>`.

```
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\CredentialSpecs
```

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

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

Example gMSA configuration

When configuring a gMSA credential spec for a service, you only need to specify a credential spec with `config`, as shown in the following example:

```
services:
 myservice:
   image: myimage:latest
   credential_spec:
     config: my_credential_spec
```

```
configs:
 my_credentials_spec:
   file: ./my-credential-spec.json|
```

depends_on

With the `depends_on` attribute, you can control the order of service startup and shutdown. It is useful if services are closely coupled, and the startup sequence impacts the application's functionality.

Short syntax

The short syntax variant only specifies service names of the dependencies. Service dependencies cause the following behaviors:

Compose creates services in dependency order. In the following example, `db` and `redis` are created before `web`.

Compose removes services in dependency order. In the following example, `web` is removed before `db` and `redis`.

Simple example:

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

Compose guarantees dependency services have been started before starting a dependent service. Compose waits for dependency services to be "ready" before starting a dependent service.

Long syntax

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

`restart`: When set to `true` Compose restarts this service after it updates the dependency service. This applies to an explicit restart controlled by a Compose operation, and excludes automated restart by the container runtime after the container dies. Introduced in Docker Compose version 2.17.0.

`condition`: Sets the condition under which dependency is considered satisfied

- `service_started`: An equivalent of the short syntax described above
- `service_healthy`: Specifies that a dependency is expected to be "healthy" (as indicated by healthcheck) before starting a dependent service.
- `service_completed_successfully`: Specifies that a dependency is expected to run to successful completion before starting a dependent service.

`required`: When set to `false` Compose only warns you when the dependency service isn't started or available. If it's not defined the default value of `required` is `true`. Introduced in Docker Compose version 2.20.0.

Service dependencies cause the following behaviors:

Compose creates services in dependency order. In the following example, `db` and `redis` are created before `web`.

Compose waits for healthchecks to pass on dependencies marked with `service_healthy`. In the following example, `db` is expected to be "healthy" before `web` is created.

Compose removes services in dependency order. In the following example, `web` is removed before `db` and `redis`.

```
services:
 web:
   build: .
   depends_on:
     db:
       condition: service_healthy
       restart: true
     redis:
       condition: service_started
 redis:
   image: redis
```

```
 db:
   image: postgres
```

Compose guarantees dependency services are started before starting a dependent service. Compose guarantees dependency services marked with `service_healthy` are "healthy" before starting a dependent service.

deploy

`deploy` specifies the configuration for the deployment and lifecycle of services, as defined in the Compose Deploy Specification.

develop

Introduced in Docker Compose version 2.22.0

`develop` specifies the development configuration for maintaining a container in sync with source, as defined in the Development Section.

device_cgroup_rules

`device_cgroup_rules` defines a list of device cgroup rules for this container. The format is the same format the Linux kernel specifies in the Control Groups Device Whitelist Controller.

```
device_cgroup_rules:
 - 'c 1:3 mr'
 - 'a 7:* rmw'
```

devices

`devices` defines a list of device mappings for created containers in the form of `HOST_PATH:CONTAINER_PATH[:CGROUP_PERMISSIONS]`.

```
devices:
 - "/dev/ttyUSB0:/dev/ttyUSB0"
 - "/dev/sda:/dev/xvda:rwm"
```

dns

`dns` defines custom DNS servers to set on the container network interface configuration. It can be a single value or a list.

```
dns: 8.8.8.8
```

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

dns_opt

`dns_opt` list custom DNS options to be passed to the containerâ■■s DNS resolver (`/etc/resolv.conf` file on Linux).

```
dns_opt:
 - use-vc
 - no-tld-query
```

dns_search

`dns_search` defines custom DNS search domains to set on container network interface configuration. It can be a single value or a list.

```
dns_search: example.com
```

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

domainname

`domainname` declares a custom domain name to use for the service container. It must be a valid RFC 1123 hostname.

entrypoint

`entrypoint` declares the default entrypoint for the service container. This overrides the `ENTRYPOINT` instruction from the service's Dockerfile.

If `entrypoint` is non-null, Compose ignores any default command from the image, for example the `CMD` instruction in the Dockerfile.

See also `command` to set or override the default command to be executed by the entrypoint process.

In its short form, the value can be defined as a string:

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

Alternatively, the value can also be a list, in a manner similar to the Dockerfile:

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

If the value is `null`, the default entrypoint from the image is used.

If the value is `[]` (empty list) or `''` (empty string), the default entrypoint declared by the image is ignored, i.e. overridden to be empty.

env_file

The `env_file` attribute is used to specify one or more files that contain environment variables to be passed to the containers.

```
env_file: .env
```

`env_file` can also be a list. The files in the list are processed from the top down. For the same variable specified in two env files, the value from the last file in the list stands.

```
env_file:
 - ./a.env
 - ./b.env
```

List elements can also be declared as a mapping, which then lets you set an additional attribute `required`. This defaults to `true`. When `required` is set to `false` and the `.env` file is missing, Compose silently ignores the entry.

```
env_file:
 - path: ./default.env
   required: true # default
 - path: ./override.env
   required: false
```

`required` attribute is available with Docker Compose version 2.24.0 or later.

Relative path are resolved from the Compose file's parent folder. As absolute paths prevent the Compose file from being portable, Compose warns you when such a path is used to set `env_file`.

Environment variables declared in the environment section override these values. This holds true even if those values are empty or undefined.

Env_file format

Each line in an `.env` file must be in `VAR[=[VAL]]` format. The following syntax rules apply:

- Lines beginning with # are processed as comments and ignored.
- Blank lines are ignored.
- Unquoted and double-quoted (") values have Interpolation applied.
- Each line represents a key-value pair. Values can optionally be quoted.
  - `VAR=VAL` -> `VAL`
  - `VAR="VAL"` -> `VAL`
  - `VAR='VAL'` -> `VAL`
- Inline comments for unquoted values must be preceded with a space.
  - `VAR=VAL # comment` -> `VAL`
  - `VAR=VAL# not a comment` -> `VAL# not a comment`
- Inline comments for quoted values must follow the closing quote.
  - `VAR="VAL # not a comment"` -> `VAL # not a comment`
  - `VAR="VAL" # comment` -> `VAL`

  Single-quoted (') values are used literally.

- `VAR='$OTHER'` -> `$OTHER`
- `VAR='${OTHER}'` -> `${OTHER}`

Quotes can be escaped with `\`.

- `VAR='Let\'s go!'` -> `Let's go!`
- `VAR="{\"hello\": \"json\"}"` -> `{"hello": "json"}`

Common shell escape sequences including `\n`, `\r`, `\t`, and `\\` are supported in double-quoted values.

- `VAR="some\tvalue"` -> `some value`
- `VAR='some\tvalue'` -> `some\tvalue`
- `VAR=some\tvalue` -> `some\tvalue`

`VAL` may be omitted, in such cases the variable value is an empty string. `=VAL` may be omitted, in such cases the variable is unset.

```
# Set Rails/Rack environment
RACK_ENV=development
VAR="quoted"
```

environment

The `environment` attribute defines environment variables set in the container. `environment` can use either an array or a map. Any boolean values; true, false, yes, no, should be enclosed in quotes to ensure they are not converted to True or False by the YAML parser.

Environment variables can be declared by a single key (no value to equals sign). In this case Compose relies on you to resolve the value. If the value is not resolved, the variable is unset and is removed from the service container environment.

Map syntax:

```
environment:
 RACK_ENV: development
 SHOW: "true"
 USER_INPUT:
```

Array syntax:

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

When both `env_file` and `environment` are set for a service, values set by `environment` have precedence.

expose

`expose` defines the (incoming) port or a range of ports that Compose exposes from the container. These ports must be accessible to linked services and should not be published to the host machine. Only the internal container ports can be specified.

Syntax is `<portnum>/[<proto>]` or `<startport-endport>/[<proto>]` for a port range. When not explicitly set, `tcp` protocol is used.

```
expose:
 - "3000"
 - "8000"
 - "8080-8085/tcp"
```

Note

If the Dockerfile for the image already exposes ports, it is visible to other containers on the network even if `expose` is not set in your Compose file.

extends

`extends` lets you share common configurations among different files, or even different projects entirely. With `extends` you can define a common set of service options in one place and refer to it from anywhere. You can refer to another Compose file and select a service you want to also use in your own application, with the ability to override some attributes for your own needs.

You can use `extends` on any service together with other configuration keys. The `extends` value must be a mapping defined with a required `service` and an optional `file` key.

```
extends:
 file: common.yml
 service: webapp
```

- `service`: Defines the name of the service being referenced as a base, for example `web` or `database`.
- `file`: The location of a Compose configuration file defining that service.

When a service uses `extends`, it can also specify dependencies on other resources, an explicit `volumes` declaration for instance. However, it's important to note that `extends` does not automatically incorporate the target volume definition into the extending Compose file. Instead, you are responsible for ensuring that an equivalent resource exists for the service being extended to maintain consistency. Docker Compose verifies that a resource with the referenced ID is present within the Compose model.

Dependencies on other resources in an `extends` target can be:

- An explicit reference by `volumes`, `networks`, `configs`, `secrets`, `links`, `volumes_from` or `depends_on`
- A reference to another service using the `service:{name}` syntax in namespace declaration (`ipc`, `pid`, `network_mode`)

Circular references with `extends` are not supported, Compose returns an error when one is detected.

Finding referenced service

`file` value can be:

- Not present. This indicates that another service within the same Compose file is being referenced.
- File path, which can be either:
  - Relative path. This path is considered as relative to the location of the main Compose file.
  - Absolute path.

A service denoted by `service` must be present in the identified referenced Compose file. Compose returns an error if:

- The service denoted by `service` is not found.
- The Compose file denoted by `file` is not found.

Merging service definitions

Two service definitions, the main one in the current Compose file and the referenced one specified by `extends`, are merged in the following way:

- Mappings: Keys in mappings of the main service definition override keys in mappings of the referenced service definition. Keys that aren't overridden are included as is.
- Sequences: Items are combined together into a new sequence. The order of elements is preserved with the referenced items coming first and main items after.
- Scalars: Keys in the main service definition take precedence over keys in the referenced one.

Mappings

The following keys should be treated as mappings: `annotations`, `build.args`, `build.labels`, `build.extra_hosts`, `deploy.labels`, `deploy.update_config`, `deploy.rollback_config`, `deploy.restart_policy`, `deploy.resources.limits`, `environment`, `healthcheck`, `labels`, `logging.options`, `sysctls`, `storage_opt`, `extra_hosts`, `ulimits`.

One exception that applies to `healthcheck` is that the main mapping cannot specify `disable: true` unless the referenced mapping also specifies `disable: true`. Compose returns an error in this case.

For example, the input below:

```
services:
 common:
   image: busybox
   environment:
     TZ: utc
     PORT: 80
 cli:
   extends:
     service: common
   environment:
     PORT: 8080
```

Produces the following configuration for the `cli` service. The same output is produced if array syntax is used.

```
environment:
 PORT: 8080
 TZ: utc
image: busybox
```

Items under `blkio_config.device_read_bps`, `blkio_config.device_read_iops`, `blkio_config.device_write_bps`, `blkio_config.device_write_iops`, `devices` and `volumes` are also treated as mappings where key is the target path inside the container.

For example, the input below:

```
services:
 common:
   image: busybox
   volumes:
     - common-volume:/var/lib/backup/data:rw
 cli:
   extends:
     service: common
   volumes:
     - cli-volume:/var/lib/backup/data:ro
```

Produces the following configuration for the `cli` service. Note that the mounted path now points to the new volume name and `ro` flag was applied.

```
image: busybox
volumes:
- cli-volume:/var/lib/backup/data:ro
```

If the referenced service definition contains `extends` mapping, the items under it are simply copied into the new merged definition. The merging process is then kicked off again until no `extends` keys are remaining.

For example, the input below:

```
services:
 base:
   image: busybox
   user: root
 common:
   image: busybox
   extends:
     service: base
 cli:
   extends:
     service: common
```

Produces the following configuration for the `cli` service. Here, `cli` services gets `user` key from `common` service, which in turn gets this key from `base` service.

```
image: busybox
user: root
```

Sequences

The following keys should be treated as sequences: `cap_add`, `cap_drop`, `configs`, `deploy.placement.constraints`, `deploy.placement.preferences`, `deploy.reservations.generic_resources`, `device_cgroup_rules`, `expose`, `external_links`, `ports`, `secrets`, `security_opt`. Any duplicates resulting from the merge are removed so that the sequence only contains unique elements.

For example, the input below:

```
services:
 common:
   image: busybox
   security_opt:
     - label:role:ROLE
 cli:
   extends:
     service: common
   security_opt:
     - label:user:USER
```

Produces the following configuration for the `cli` service.

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

In case list syntax is used, the following keys should also be treated as sequences: `dns`, `dns_search`, `env_file`, `tmpfs`. Unlike sequence fields mentioned above, duplicates resulting from the merge are not removed.

Scalars

Any other allowed keys in the service definition should be treated as scalars.

external_links

`external_links` link service containers to services managed outside of your Compose application. `external_links` define the name of an existing service to retrieve using the platform lookup mechanism. An alias of the form `SERVICE:ALIAS` can be specified.

```
external_links:
 - redis
 - database:mysql
 - database:postgresql
```

extra_hosts

`extra_hosts` adds hostname mappings to the container network interface configuration (`/etc/hosts` for Linux).

Short syntax

Short syntax uses plain strings in a list. Values must set hostname and IP address for additional hosts in the form of `HOSTNAME=IP`.

```
extra_hosts:
 - "somehost=162.242.195.82"
 - "otherhost=50.31.209.229"
 - "myhostv6=::1"
```

IPv6 addresses can be enclosed in square brackets, for example:

```
extra_hosts:
 - "myhostv6=[::1]"
```

The separator = is preferred, but `:` can also be used. Introduced in Docker Compose version 2.24.1. For example:

```
extra_hosts:
 - "somehost:162.242.195.82"
 - "myhostv6:::1"
```

Long syntax

Alternatively, `extra_hosts` can be set as a mapping between hostname(s) and IP(s)

```
extra_hosts:
 somehost: "162.242.195.82"
 otherhost: "50.31.209.229"
 myhostv6: "::1"
```

Compose creates a matching entry with the IP address and hostname in the container's network configuration, which means for Linux `/etc/hosts` get extra lines:

```
162.242.195.82  somehost
50.31.209.229   otherhost
::1             myhostv6
```

group_add

`group_add` specifies additional groups, by name or number, which the user inside the container must be a member of.

An example of where this is useful is when multiple containers (running as different users) need to all read or write the same file on a shared volume. That file can be owned by a group shared by all the containers, and specified in `group_add`.

```
services:
 myservice:
   image: alpine
   group_add:
     - mail
```

Running `id` inside the created container must show that the user belongs to the `mail` group, which would not have been the case if `group_add` were not declared.

healthcheck

The `healthcheck` attribute declares a check that's run to determine whether or not the service containers are "healthy". It works in the same way, and has the same default values, as the HEALTHCHECK Dockerfile instruction set by the service's Docker image. Your Compose file can override the values set in the Dockerfile.

For more information on `HEALTHCHECK`, see the Dockerfile reference.

```
healthcheck:
 test: ["CMD", "curl", "-f", "http://localhost"]
 interval: 1m30s
 timeout: 10s
 retries: 3
 start_period: 40s
 start_interval: 5s
```

`interval`, `timeout`, `start_period`, and `start_interval` are specified as durations. Introduced in Docker Compose version 2.20.2

`test` defines the command Compose runs to check container health. It can 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"]
```

Using `CMD-SHELL` runs the command configured as a string using the container's default shell (`/bin/sh` for Linux). Both forms below are equivalent:

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

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

`NONE` disables the healthcheck, and is mostly useful to disable the Healthcheck Dockerfile instruction set by the service's Docker image. Alternatively, the healthcheck set by the image can be disabled by setting `disable: true`:

```
healthcheck:
 disable: true
```

hostname

`hostname` declares a custom host name to use for the service container. It must be a valid RFC 1123 hostname.

image

`image` specifies the image to start the container from. `image` must follow the Open Container Specification addressable image format, as `[<registry>/][<project>/]<image>[:<tag>|@<digest>]`.

```
    image: redis
   image: redis:5
   image: redis@sha256:0ed5d5928d4737458944eb604cc8509e245c3e19d02ad83935398bc4b991aac7
   image: library/redis
   image: docker.io/library/redis
   image: my_private.registry:5000/redis
```

If the image does not exist on the platform, Compose attempts to pull it based on the `pull_policy`. If you are also using the Compose Build Specification, there are alternative options for controlling the precedence of pull over building the image from source, however pulling the image is the default behavior.

`image` may be omitted from a Compose file as long as a `build` section is declared. If you are not using the Compose Build Specification, Compose won't work if `image` is missing from the Compose file.

init

`init` runs an init process (PID 1) inside the container that forwards signals and reaps processes. Set this option to `true` to enable this feature for the service.

```
services:
 web:
   image: alpine:latest
   init: true
```

The init binary that is used is platform specific.

ipc

`ipc` configures the IPC isolation mode set by the service container.

- `shareable`: Gives the container its own private IPC namespace, with a possibility to share it with other containers.
- `service:{name}`: Makes the container join another container's (`shareable`) IPC namespace.

```
 ipc: "shareable"
ipc: "service:[service name]"
```

isolation

`isolation` specifies a containerâ■■s isolation technology. Supported values are platform specific.

labels

`labels` add metadata to containers. You can use either an array or a map.

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"
```

Compose creates containers with canonical labels:

- `com.docker.compose.project` set on all resources created by Compose to the user project name
- `com.docker.compose.service` set on service containers with service name as defined in the Compose file

The `com.docker.compose` label prefix is reserved. Specifying labels with this prefix in the Compose file results in a runtime error.

links

`links` defines a network 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 is specified.

Links are not required to enable services to communicate. When no specific network configuration is set, any service is able to reach any other service at that serviceâ■■s name on the `default` network. If services do declare networks they are attached to, `links` does not override the network configuration and services not attached to a shared network are not be able to communicate. Compose doesn't warn you about a configuration mismatch.

Links also express implicit dependency between services in the same way as depends_on, so they determine the order of service startup.

logging

`logging` defines the 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. The default and available values are platform specific. Driver specific options can be set with `options` as key-value pairs.

mac_address

Available with Docker Compose version 2.24.0 and later.

`mac_address` sets a MAC address for the service container.

> **Note** Container runtimes might reject this value (ie. Docker Engine >= v25.0). In that case, you should use networks.mac_address instead.

mem_limit

`mem_limit` configures a limit on the amount of memory a container can allocate, set as a string expressing a byte value.

When set, `mem_limit` must be consistent with the `limits.memory` attribute in the Deploy Specification.

mem_reservation

`mem_reservation` configures a reservation on the amount of memory a container can allocate, set as a string expressing a byte value.

When set, `mem_reservation` must be consistent with the `reservations.memory` attribute in the Deploy Specification.

mem_swappiness

`mem_swappiness` defines as a percentage, a value between 0 and 100, for the host kernel to swap out anonymous memory pages used by a container.

- `0`: Turns off anonymous page swapping.
- `100`: Sets all anonymous pages as swappable.

The default value is platform specific.

memswap_limit

`memswap_limit` defines the amount of memory the container is allowed to swap to disk. This is a modifier attribute that only has meaning if `memory` is also set. Using swap lets the container write excess memory requirements to disk when the container has exhausted all the memory that is available to it. There is a performance penalty for applications that swap memory to disk often.

- If `memswap_limit` is set to a positive integer, then both `memory` and `memswap_limit` must be set. `memswap_limit` represents the total amount of memory and swap that can be used, and `memory` controls the amount used by non-swap memory. So if `memory="300m"` and `memswap_limit="1g"`, the container can use 300m of memory and 700m (1g - 300m) swap.
- If `memswap_limit` is set to 0, the setting is ignored, and the value is treated as unset.
- If `memswap_limit` is set to the same value as `memory`, and `memory` is set to a positive integer, the container does not have access to swap.
- If `memswap_limit` is unset, and `memory` is set, the container can use as much swap as the `memory` setting, if the host container has swap memory configured. For instance, if `memory="300m"` and `memswap_limit` is not set, the container can use 600m in total of memory and swap.
- If `memswap_limit` is explicitly set to -1, the container is allowed to use unlimited swap, up to the amount available on the host system.

network_mode

`network_mode` sets a service container's network mode.

- `none`: Turns off all container networking.
- `host`: Gives the container raw access to the host's network interface.
- `service:{name}`: Gives the containers access to the specified service only. For more information, see Container networks.

```
 network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
```

When set, the `networks` attribute is not allowed and Compose rejects any Compose file containing both attributes.

networks

The `networks` attribute defines the networks that service containers are attached to, referencing entries under the `networks` top-level element. The `networks` attribute helps manage the networking aspects of containers, providing control over how services are segmented and interact within the Docker environment. This is used to specify which networks the containers for that service should connect to. This is important for defining how containers communicate with each other and externally.

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

For more information about the `networks` top-level element, see Networks.

aliases

`aliases` declares alternative hostnames for the service on the network. Other containers on the same network can use either the service name or an alias to connect to one of the service's containers.

Since `aliases` are 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.

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

In the following example, service `frontend` is able to reach the `backend` service at the hostname `backend` or `database` on the `back-tier` network. The service `monitoring` is able to reach same `backend` service at `backend` or `mysql` on the `admin` network.

```
services:
 frontend:
   image: example/webapp
   networks:
     - front-tier
     - back-tier

 monitoring:
   image: example/monitoring
   networks:
     - admin

 backend:
   image: example/backend
   networks:
     back-tier:
       aliases:
         - database
     admin:
       aliases:
         - mysql

networks:
 front-tier:
 back-tier:
 admin:
```

ipv4_address, ipv6_address

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

The corresponding network configuration in the top-level networks section must have an `ipam` attribute with subnet configurations covering each static address.

```
services:
 frontend:
   image: example/webapp
   networks:
     front-tier:
       ipv4_address: 172.16.238.10
       ipv6_address: 2001:3984:3989::10

networks:
 front-tier:
   ipam:
     driver: default
     config:
```

```
      - subnet: "172.16.238.0/24"
      - subnet: "2001:3984:3989::/64"
```

link_local_ips

`link_local_ips` specifies a list of link-local IPs. Link-local IPs are special IPs which belong to a well known subnet and are purely managed by the operator, usually dependent on the architecture where they are deployed.

Example:

```
services:
 app:
   image: busybox
   command: top
   networks:
     app_net:
       link_local_ips:
         - 57.123.22.11
         - 57.123.22.13
networks:
 app_net:
   driver: bridge
```

mac_address

Introduced in Docker Compose version 2.23.2

`mac_address` sets the MAC address used by the service container when connecting to this particular network.

priority

`priority` indicates in which order Compose connects the serviceâ■■s containers to its networks. If unspecified, the default value is 0.

In the following example, the app service connects to `app_net_1` first as it has the highest priority. It then connects to `app_net_3`, then `app_net_2`, which uses the default priority value of 0.

```
services:
 app:
   image: busybox
   command: top
   networks:
     app_net_1:
       priority: 1000
     app_net_2:

     app_net_3:
       priority: 100
networks:
 app_net_1:
 app_net_2:
 app_net_3:
```

oom_kill_disable

If `oom_kill_disable` is set, Compose configures the platform so it won't kill the container in case of memory starvation.

oom_score_adj

`oom_score_adj` tunes the preference for containers to be killed by platform in case of memory starvation. Value must be within -1000,1000 range.

pid

`pid` sets the PID mode for container created by Compose. Supported values are platform specific.

pids_limit

`pids_limit` tunes a containerâ■■s PIDs limit. Set to -1 for unlimited PIDs.

```
pids_limit: 10
```

When set, `pids_limit` must be consistent with the `pids` attribute in the Deploy Specification.

platform

`platform` defines the target platform the containers for the service run on. It uses the `os[/arch[/variant]]` syntax.

The values of `os`, `arch`, and `variant` must conform to the convention used by the OCI Image Spec.

Compose uses this attribute to determine which version of the image is pulled and/or on which platform the serviceâ■■s build is performed.

```
platform: darwin
platform: windows/amd64
platform: linux/arm64/v8
```

ports

The `ports` is used to define the port mappings between the host machine and the containers. This is crucial for allowing external access to services running inside containers. It can be defined using short syntax for simple port mapping or long syntax, which includes additional options like protocol type and network mode.

> **Note**
>
> Port mapping must not be used with `network_mode: host` otherwise a runtime error occurs.

Short syntax

The short syntax is a colon-separated string to set the host IP, host port, and container port in the form:

`[HOST:]CONTAINER[/PROTOCOL]` where:

- `HOST` is `[IP:](port | range)`
- `CONTAINER` is `port | range`
- `PROTOCOL` to restrict port to specified protocol. `tcp` and `udp` values are defined by the Specification, Compose offers support for platform-specific protocol names.

If host IP is not set, it binds to all network interfaces. Ports can be either a single value or a range. Host and container must use equivalent ranges.

Either specify both ports (`HOST:CONTAINER`), or just the container port. In the latter case, the container runtime automatically allocates any unassigned port of the host.

`HOST:CONTAINER` should always be specified as a (quoted) string, to avoid conflicts with yaml base-60 float.

Examples:

```
ports:
 - "3000"
 - "3000-3005"
 - "8000:8000"
 - "9090-9091:8080-8081"
 - "49100:22"
 - "8000-9000:80"
 - "127.0.0.1:8001:8001"
 - "127.0.0.1:5000-5010:5000-5010"
 - "6060:6060/udp"
```

> **Note**
>
> If Host IP mapping is not supported by a container engine, Compose rejects the Compose file and ignores the specified host IP.

Long syntax

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

- `target`: The container port
- `published`: The publicly exposed port. It is defined as a string and can be set as a range using syntax `start-end`. It means the actual port is assigned a remaining available port, within the set range.
- `host_ip`: The Host IP mapping, unspecified means all network interfaces (`0.0.0.0`).
- `protocol`: The port protocol (`tcp` or `udp`). Defaults to `tcp`.
- `app_protocol`: The application protocol (TCP/IP level 4 / OSI level 7) this port is used for. This is optional and can be used as a hint for Compose to offer richer behavior for protocols that it understands. Introduced in Docker Compose version 2.26.0.

- `mode`: `host`: For publishing a host port on each node, or `ingress` for a port to be load balanced. Defaults to `ingress`.
- `name`: A human-readable name for the port, used to document it's usage within the service.

```
ports:
 - name: web
   target: 80
   host_ip: 127.0.0.1
   published: "8080"
   protocol: tcp
   app_protocol: http
   mode: host

 - name: web-secured
   target: 443
   host_ip: 127.0.0.1
   published: "8083-9000"
   protocol: tcp
   app_protocol: https
   mode: host
```

privileged

`privileged` configures the service container to run with elevated privileges. Support and actual impacts are platform specific.

profiles

`profiles` defines a list of named profiles for the service to be enabled under. If unassigned, the service is always started but if assigned, it is only started if the profile is activated.

If present, `profiles` follow the regex format of `[a-zA-Z0-9][a-zA-Z0-9_.-]+`.

```
services:
 frontend:
   image: frontend
   profiles: ["frontend"]

 phpmyadmin:
   image: phpmyadmin
   depends_on:
     - db
   profiles:
     - debug
```

pull_policy

`pull_policy` defines the decisions Compose makes when it starts to pull images. Possible values are:

- `always`: Compose always pulls the image from the registry.
- `never`: Compose doesn't pull the image from a registry and relies on the platform cached image. If there is no cached image, a failure is reported.
- `missing`: Compose pulls the image only if it's not available in the platform cache. This is the default option if you are not also using the Compose Build Specification. `if_not_present` is considered an alias for this value for backward compatibility.
- `build`: Compose builds the image. Compose rebuilds the image if it's already present.

read_only

`read_only` configures the service container to be created with a read-only filesystem.

restart

`restart` defines the policy that the platform applies on container termination.

- `no`: The default restart policy. It does not restart the container under any circumstances.
- `always`: The policy always restarts the container until its removal.
- `on-failure[:max-retries]`: The policy restarts the container if the exit code indicates an error. Optionally, limit the number of restart retries the Docker daemon attempts.
- `unless-stopped`: The policy restarts the container irrespective of the exit code but stops restarting when the service is stopped or removed.

```
 restart: "no"
restart: always
restart: on-failure
restart: on-failure:3
restart: unless-stopped
```

You can find more detailed information on restart policies in the Restart Policies (--restart) section of the Docker run reference page.

runtime

`runtime` specifies which runtime to use for the serviceâ■■s containers.

For example, `runtime` can be the name of an implementation of OCI Runtime Spec, such as "runc".

```
web:
 image: busybox:latest
 command: true
 runtime: runc
```

The default is `runc`. To use a different runtime, see Alternative runtimes.

scale

`scale` specifies the default number of containers to deploy for this service. When both are set, `scale` must be consistent with the `replicas` attribute in the Deploy Specification.

secrets

The `secrets` attribute grants access to sensitive data defined by the secrets top-level element on a per-service basis. Services can be granted access to multiple secrets.

Two different syntax variants are supported; the short syntax and the long syntax. Long and short syntax for secrets may be used in the same Compose file.

Compose reports an error if the secret doesn't exist on the platform or isn't defined in the `secrets` top-level section of the Compose file.

Defining a secret in the top-level `secrets` must not imply granting any service access to it. Such grant must be explicit within service specification as secrets service element.

Short syntax

The short syntax variant only specifies the secret name. This grants the container access to the secret and mounts it as read-only to `/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 `frontend` service access to the `server-certificate` secret. The value of `server-certificate` is set to the contents of the file `./server.cert`.

```
services:
 frontend:
   image: example/webapp
   secrets:
     - server-certificate
secrets:
 server-certificate:
   file: ./server.cert
```

Long syntax

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

- `source`: The name of the secret as it exists on the platform.
- `target`: The name of the file to be mounted in `/run/secrets/` in the service's task container, or absolute path of the file if an alternate location is required. 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. Default value is USER running container.
- `mode`: The permissions for the file to be mounted in `/run/secrets/` in the service's task containers, in octal notation. The default value is world-readable permissions (mode `0444`). The writable bit must be ignored if set. The executable bit may be set.

The following example sets the name of the `server-certificate` secret file to `server.cert` within the container, sets the mode to `0440` (group-readable), and sets the user and group to `103`. The value of `server-certificate` is set to the contents of the file `./server.cert`.

```
services:
 frontend:
   image: example/webapp
   secrets:
     - source: server-certificate
       target: server.cert
       uid: "103"
       gid: "103"
       mode: 0440
secrets:
 server-certificate:
   file: ./server.cert
```

security_opt

`security_opt` overrides the default labeling scheme for each container.

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

For further default labeling schemes you can override, see Security configuration.

shm_size

`shm_size` configures the size of the shared memory (`/dev/shm` partition on Linux) allowed by the service container. It's specified as a byte value.

stdin_open

`stdin_open` configures a service's container to run with an allocated stdin. This is the same as running a container with the `-i` flag. For more information, see Keep STDIN open.

Supported values are `true` or `false`.

stop_grace_period

`stop_grace_period` specifies how long Compose must wait when attempting to stop a container if it doesn't handle SIGTERM (or whichever stop signal has been specified with `stop_signal`), before sending SIGKILL. It's specified as a duration.

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

Default value is 10 seconds for the container to exit before sending SIGKILL.

stop_signal

`stop_signal` defines the signal that Compose uses to stop the service containers. If unset containers are stopped by Compose by sending `SIGTERM`.

```
stop_signal: SIGUSR1
```

storage_opt

`storage_opt` defines storage driver options for a service.

```
storage_opt:
 size: '1G'
```

sysctls

`sysctls` defines kernel parameters to set in the container. `sysctls` can use either an array or a map.

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

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

You can only use sysctls that are namespaced in the kernel. Docker does not support changing sysctls inside a container that also modify the host system. For an overview of supported sysctls, refer to configure namespaced kernel parameters (sysctls) at runtime.

tmpfs

`tmpfs` mounts a temporary file system inside the container. It can be a single value or a list.

```
tmpfs: /run
```

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

tty

`tty` configures a service's container to run with a TTY. This is the same as This is the same as running a container with the `-t` or `--tty` flag. For more information, see Allocate a pseudo-TTY.

Supported values are `true` or `false`.

ulimits

`ulimits` overrides the default ulimits for a container. It's specified either as an integer for a single limit or as mapping for soft/hard limits.

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

user

`user` overrides the user used to run the container process. The default is set by the image (i.e. Dockerfile USER). If it's not set, then `root`.

userns_mode

`userns_mode` sets the user namespace for the service. Supported values are platform specific and may depend on platform configuration.

```
userns_mode: "host"
```

uts

Introduced in Docker Compose version 2.15.1

`uts` configures the UTS namespace mode set for the service container. When unspecified it is the runtime's decision to assign a UTS namespace, if supported. Available values are:

- `'host'`: Results in the container using the same UTS namespace as the host.

  ```
  uts: "host"
  ```

volumes

The `volumes` attribute define mount host paths or named volumes that are accessible by service containers. You can use `volumes` to define multiple types of mounts; `volume`, `bind`, `tmpfs`, or `npipe`.

If the mount is a host path and is only used by a single service, it can be declared as part of the service definition. To reuse a volume across multiple services, a named volume must be declared in the `volumes` top-level element.

The following example shows a named volume (`db-data`) being used by the `backend` service, and a bind mount defined for a single service.

```
services:
 backend:
   image: example/backend
   volumes:
     - type: volume
       source: db-data
       target: /data
       volume:
```

```
        nocopy: true
        subpath: sub
    - type: bind
      source: /var/run/postgres/postgres.sock
      target: /var/run/postgres/postgres.sock

volumes:
 db-data:
```

For more information about the `volumes` top-level element, see Volumes.

Short syntax

The short syntax uses a single string with colon-separated values to specify a volume mount (`VOLUME:CONTAINER_PATH`), or an access mode (`VOLUME:CONTAINER_PATH:ACCESS_MODE`).

- `VOLUME`: Can be either a host path on the platform hosting containers (bind mount) or a volume name.
- `CONTAINER_PATH`: The path in the container where the volume is mounted.

  `ACCESS_MODE`: A comma-separated `,` list of options:
  - `rw`: Read and write access. This is the default if none is specified.
  - `ro`: Read-only access.
  - `z`: SELinux option indicating that the bind mount host content is shared among multiple containers.
  - `Z`: SELinux option indicating that the bind mount host content is private and unshared for other containers.

  Note

  The SELinux re-labeling bind mount option is ignored on platforms without SELinux.

  Note Relative host paths are only supported by Compose that deploy to a local container runtime. This is because the relative path is resolved from the Compose fileâ■■s parent directory which is only applicable in the local case. When Compose deploys to a non-local platform it rejects Compose files which use relative host paths with an error. To avoid ambiguities with named volumes, relative paths should always begin with `.` or `...`

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. Either `volume`, `bind`, `tmpfs`, `npipe`, or `cluster`
- `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. 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`: Used to configure additional bind options:
  - `propagation`: The propagation mode used for the bind.
  - `create_host_path`: Creates a directory at the source path on host if there is nothing present. Compose does nothing if there is something present at the path. This is automatically implied by short syntax for backward compatibility with `docker-compose` legacy.
  - `selinux`: The SELinux re-labeling option `z` (shared) or `Z` (private)

  `volume`: Configures additional volume options:
  - `nocopy`: Flag to disable copying of data from a container when a volume is created.
  - `subpath`: Path inside a volume to mount instead of the volume root.

  `tmpfs`: Configures additional tmpfs options:
  - `size`: The size for the tmpfs mount in bytes (either numeric or as bytes unit).
  - `mode`: The file mode for the tmpfs mount as Unix permission bits as an octal number. Introduced in Docker Compose version 2.14.0.
- `consistency`: The consistency requirements of the mount. Available values are platform specific.

  Tip

  Working with large repositories or monorepos, or with virtual file systems that are no longer scaling with your codebase? Compose now takes advantage of Synchronized file shares and automatically creates file shares for bind mounts. Ensure you're signed in to Docker with a paid subscription and have enabled both **Access experimental features** and **Manage Synchronized file shares with Compose** in Docker Desktop's settings.

volumes_from

`volumes_from` mounts all of the volumes from another service or container. You can optionally specify read-only access `ro` or read-write `rw`. If no access level is specified, then read-write access is used.

You can also mount volumes from a container that is not managed by Compose by using the `container:` prefix.

```
volumes_from:
 - service_name
 - service_name:ro
 - container:container_name
 - container:container_name:rw
```

working_dir

`working_dir` overrides the container's working directory which is specified by the image, for example Dockerfile's `WORKDIR`.