Docker-Compose: le fichier YAML version 3

Table of Contents

        - networks
        - alias
        - ipv4_address, ipv6_address
        - pid
        - ports
            - Syntaxe courte
            - Syntaxe longue
        - restart
        - secrets
            - Syntaxe courte
            - Syntaxe longue
        - security_opt
        - stop_grace_period
        - stop_signal
        - sysctls
        - tmpfs
            - version 2 et plus.
            - version 3.6 et plus
        - ulimits
        - userns_mode
        - volumes
            - Syntaxe courte
            - Syntaxe longue
            - Volumes de services, swarms et fichiers de pile
        - domainname, hostname, ipc, mac_address, privileged, read_only, shm_size, stdin_open, tty, user, working_dir
- Référence de configuration de volume
            - driver
            - driver_opts
            - external
            - labels
            - name
- Référence de configuration du réseau
        - driver
            - bridge
            - overlay
            - host ou none
            - driver_opts
            - attachable
            - enable_ipv6
            - ipam
            - internal
            - labels
- external
            - name

Le fichier Compose est un fichier YAML définissant les services, les réseaux et les volumes. Le chemin par défaut pour un fichier Compose est ./docker-compose.yml.

On peut utiliser une extension **.yml** ou **.yaml** pour ce fichier. Ils travaillent tous les deux.

Ces rubriques décrivent la version 3 du format de fichier Compose. Ceci est la dernière version.

Les rubriques de cette page de référence sont organisées en ordre alphabétique par clé de niveau supérieur afin de refléter la structure du fichier Compose lui-même. Les clés de niveau supérieur qui définissent une section dans le fichier de configuration, telles que **build**, **deploy**, **depends_on**, **networks**, etc., sont répertoriées avec les options qu'ils prennent en charge . Cela correspond à la structure d'indentation `<clé>:<option>:<valeur>` du fichier Compose.

Le didacticiel de prise en main, qui utilise la version 3, compose des fichiers de pile pour implémenter des applications multi-conteneurs, des définitions de service et le mode swarm.

Syntaxe des métriques

Spécifier les durées

Certaines options de configuration, telles que les sous-options `interval` et `timeout` pour `check`, acceptent une durée sous la forme d'une chaîne dans un format ressemblant à ceci:

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

Les unités prises en charge sont `us, ms, s, m et h.`

Spécifier des valeurs d'octet

Certaines options de configuration, telles que la sous-option `shm_size` pour `build`, acceptent une valeur d'octet sous forme de chaîne dans un format ressemblant à ceci:

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

Les unités prises en charge sont `b, k, m et g` et leur notation alternative `kb, mb et gb`. Les valeurs décimales ne sont pas prises en charge pour le moment.

Références de la Configuration de service

Une définition de service contient une configuration qui est appliquée à chaque conteneur démarré pour ce service, un peu comme si on transmettait des paramètres de ligne de commande à un conteneur de docker. De même, les définitions de réseau et de volume sont analogues à la création réseau de docker et à la création de volume de docker.

Comme avec `docker container create`, les options spécifiées dans le fichier Docker, telles que `CMD`, `EXPOSE`, `VOLUME`, `ENV`, sont respectées par défaut. Il n'est pas besoin de les spécifier à nouveau dans docker-compose.yml.

On peut utiliser des variables d'environnement dans les valeurs de configuration avec une syntaxe de type `${Variable}` - semblable à Bash.

Cette section contient une liste de toutes les options de configuration prises en charge par une définition de service dans la version 3.

build

Syntaxe

Options de configuration appliquées au moment de la construction.

build peut être spécifié soit en tant que chaîne contenant un chemin d'accès au contexte de construction:

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

Ou, en tant qu'objet avec le chemin spécifié sous contexte et éventuellement Dockerfile and args:

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

En vous spécifiant `image` ainsi que `build`, Compose nomme l'image construite avec l'application Webapp et la balise facultative spécifiée dans l'image:

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

Cela donne une image nommée webapp et tag tag, construite à partir de ./dir.

cette option est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3). La commande docker stack accepte uniquement les images prédéfinies.

Options disponibles

context

Soit un chemin d'accès à un répertoire contenant un fichier Docker, soit une URL vers un référentiel git.

Lorsque la valeur fournie est un chemin relatif, elle est interprétée comme relative à l'emplacement du fichier Compose. Ce répertoire est également le contexte de construction envoyé au démon Docker.

Compose le construit et le marque avec un nom généré, puis utilise cette image.

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

dockerfile

En indiquant `Dockerfile-alternative` Compose utilise un autre fichier pour construire. Un chemin de construction doit également être spécifié.

```

```

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

args

Pour ajouter des arguments de construction, qui sont des variables d'environnement accessibles uniquement pendant le processus de construction.

Tout d'abord, spécifier les arguments dans le fichier Docker:

```
ARG buildno
ARG gitcommithash

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

Puis spécifier les arguments sous la clé `build`. On peut passer un mapping ou une liste:

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

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

Dans le fichier Docker, si on spécifie ARG avant l'instruction FROM, ARG n'est pas disponible dans les instructions de construction sous FROM. Pour qu'un argument soit disponible aux deux emplacements, il faut également le spécifier dans l'instruction FROM.

On peut omettre la valeur lors de la spécification d'un argument de construction. Dans ce cas, sa valeur au moment de la construction est la valeur de l'environnement dans lequel Compose est en cours d'exécution.

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

les valeurs booléennes YAML (true, false, yes, no, on, off) doivent être placées entre guillemets, afin que l'analyseur les interprète comme des chaînes.

cache_from

```
nouvelle option dans la v3.2
```

Liste des images utilisées par le moteur pour la résolution du cache.

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

labels

```
nouvelle option dans la v3.3
```

Ajoute des métadonnées à l'image obtenue à l'aide des étiquettes Docker.

On peut utiliser un tableau ou un dictionnaire.

On recommande d'utiliser la notation DNS inversée pour éviter que les étiquettes ne soient en conflit avec celles utilisées par d'autres logiciels.

```
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

```

```

```
nouvelle option dans la version 3.5
```

Définit la taille de la partition /dev/shm pour les conteneurs de cette construction. Spécifier une valeur entière représentant le nombre d'octets ou une chaîne exprimant une valeur d'octet.

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

build:
  context: .
  shm_size: 10000000
```

target

```
nouvelle option dans la version 3.4
```

Construire l'étape spécifiée telle que définie dans le fichier Dockerfile. Voir la documentation de construction en plusieurs étapes pour plus de détails.

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

cap_add, cap_drop

Ajouter ou supprimer des capacités de conteneur. .

```
cap_add:
  - ALL

cap_drop:
  - NET_ADMIN
  - SYS_ADMIN
```

ces options sont ignorées lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3).

cgroup_parent

Spécifie un groupe de contrôle parent facultatif pour le conteneur.

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

cette option est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3).

command

Remplace la commande par défaut.

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

La commande peut également être spécifiée en liste, comme dans dockerfile:

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

configs

Pour accorder l'accès à la configuration par service à l'aide de la configuration des configurations par service.

Deux variantes de syntaxe différentes sont prises en charge.

La configuration doit déjà exister ou être définie dans la configuration de configuration de niveau supérieur de ce fichier de pile, sinon le déploiement de la pile échoue.

Syntaxe courte

La variante de syntaxe courte spécifie uniquement le nom de configuration. Cela accorde au conteneur l'accès à la configuration et le monte à `/<nom_conf>` dans le conteneur. Le nom source et le point de montage de destination sont tous deux définis sur le nom de configuration.

L'exemple suivant utilise la syntaxe abrégée pour accorder au service Redis l'accès aux configs

my_config et my_other_config. La valeur de my_config est définie sur le contenu du fichier ./my_config.txt, et my_other_config est défini en tant que ressource externe, ce qui signifie qu'il a déjà été défini dans Docker, en exécutant la commande **docker config create** ou par une autre pile. déploiement. Si la configuration externe n'existe pas, le déploiement de la pile échoue avec une erreur de configuration introuvable.

les définitions de configuration ne sont prises en charge que dans les versions 3.3 et supérieures du format de fichier composé.

```
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
```

Syntaxe longue

La syntaxe longue fournit plus de précision dans la création de la configuration dans les conteneurs de tâches du service.

- **source**: nom de la configuration telle qu'elle existe dans Docker.
- **cible**: chemin d'accès et nom du fichier à monter dans les conteneurs de tâches du service. La valeur par défaut est /<source> si non spécifié.
- **uid et gid**: UID ou GID numérique qui possède le fichier de configuration monté dans les conteneurs de tâches du service. Les deux valeurs par défaut sont 0 sur Linux si elles ne sont pas spécifiées. Non pris en charge sous Windows.
- **mode**: autorisations du fichier monté dans les conteneurs de tâches du service, en notation octale. Par exemple, 0444 représente le monde lecture sule. La valeur par défaut est 0444. Les configurations ne peuvent pas être insérées en écriture car elles sont montées dans un système de fichiers temporaire. Par conséquent, l'orsqu'on définit le bit en écriture, celui-ci est ignoré. Le bit exécutable peut être défini.

L'exemple suivant définit le nom de my_config sur redis_config dans le conteneur, définit le mode sur 0440 (lisible par un groupe) et définit l'utilisateur et le groupe sur 103. Le service Redis n'a pas accès à la configuration my_other_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
```

On peut accorder à un service l'accès à plusieurs configurations et mélanger des syntaxes longues et courtes. Définir une configuration n'implique pas l'octroi d'un accès à un service.

container_name

Spécifiez un nom de conteneur personnalisé, plutôt qu'un nom par défaut généré.

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

Les noms de conteneur Docker doivent être uniques, on ne peut pas mettre à l'échelle un service au-delà d'un conteneur si vous avez spécifié un nom personnalisé. Tenter de le faire entraîne une erreur.

cette option est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3).

credential_spec

cette option a été ajoutée dans la v3.3.

Configure les informations d'identification pour le compte de service géré. Cette option est uniquement utilisée pour les services utilisant des conteneurs Windows. `Credential_spec` doit être au format `file://<nom de fichier> ou registry://<nom-valeur>`.

Lors de l'utilisation de fichier:, le fichier référencé doit être présent dans le sous-répertoire CredentialSpecs du répertoire de données Docker, qui est défini par défaut sur `C:\ProgramData\Docker\` sous Windows. L'exemple suivant charge les spécifications d'identification à partir d'un fichier nommé `C:\ProgramData\Docker\CredentialSpecs\my-credential-spec.json`:

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

Lorsqu'on utilise `registry:`, la spécification des informations d'identification est lue à partir du registre Windows de l'hôte du démon. Une valeur de registre avec le nom donné doit être située dans:

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

L'exemple suivant charge la spécification d'identification à partir d'une valeur nommée `my-credential-spec` dans le registre:

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

depends_on

Dépendance explicite entre services, les dépendances de service provoquent les comportements suivants:

- `docker-compose up` démarre les services dans l'ordre des dépendances. Dans l'exemple suivant, `db` et `redis` sont démarrés avant `web`.
- `docker-compose up SERVICE` inclut automatiquement les dépendances de `SERVICE`. Dans l'exemple suivant, `docker-compose up web` crée et démarre également `db` et `redis`.
- `docker-compose stop` arrête les services en ordre de dépendance. Dans l'exemple suivant, Web est arrêté avant `db` et `redis`.

Exemple simple:

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

Il y a plusieurs choses à prendre en compte lors de l'utilisation de `depend_on`:

- `depend_on` n'attend pas que `db` et `redis` soient «prêtes» avant de démarrer `Web` - mais uniquement jusqu'à ce qu'elles aient été démarrées. Pour attendre qu'un service soit prêt, il faut contrôler l'ordre de démarrage.
- La version 3 ne prend plus en charge la forme de condition de `goes_on`.
- L'option `depend_on` est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose version 3.

deploy

Version 3 seulement.

Syntaxe

Spécifie la configuration liée au déploiement et à l'exécution des services. Cela ne prend effet que lors du déploiement sur un essaim (swarn) avec déploiement de la pile de docker, et est ignoré par `docker-compose up` et `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
```

Options disponibles

endpoint_mode

```
Version 3.3 seulement.
```

Spécifie une méthode de découverte de service pour les clients externes se connectant à un essaim.

- `endpoint_mode: vip` - Docker attribue au service une adresse IP virtuelle (VIP) qui sert d'interface frontale aux clients pour qu'ils atteignent le service sur un réseau. Docker achemine les demandes entre le client et les noeuds de travail disponibles pour le service, sans que le client sache combien de noeuds participent au service, ni leurs adresses IP ou leurs ports. (Ceci est la valeur par défaut.)
- `endpoint_mode: dnsrr` - La découverte du service DNS à tour de rôle (DNSRR) n'utilise pas

une adresse IP virtuelle unique. Docker configure les entrées DNS pour le service de sorte qu'une requête DNS pour le nom du service retourne une liste d'adresses IP et que le client se connecte directement à l'une d'entre elles. Le round-robin DNS est utile dans les cas où l'on souhaite utiliser son propre équilibreur de charge ou pour des applications hybrides Windows et Linux.

```
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:
```

Les options `endpoint_mode` fonctionnent également comme des indicateurs lors de la création du service de docker de commandes CLI en mode swarm.

labels

Spécifie les étiquettes pour le service. Ces étiquettes ne sont définies que sur le service et non sur les conteneurs du service.

```
version: "3.7"
services:
  web:
```

```
    image: web
    deploy:
      labels:
        com.example.description: "Cette étiquette apparaîtra sur le service
Web"
```

Pour définir des étiquettes sur des conteneurs, utiliser la clé `labels` en dehors de deploy:

```
version: "3.7"
services:
  web:
    image: web
    labels:
      com.example.description: "Cette étiquette apparaîtra sur tous les
conteneurs du service Web"
```

mode

Soit `global` (exactement un conteneur par nœud Swarm) ou `replicated` (un nombre spécifié de conteneurs). Le défaut est `replicated`.

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

placement

Spécifie l'emplacement des contraintes et des préférences.

```
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

Si le service est répliqué (ce qui est la valeur par défaut), spécifie le nombre de conteneurs à exécuter à un moment donné.

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

ressources

Configure les contraintes de ressources.

Ceci remplace les anciennes options de contrainte de ressources pour le mode non-sarm dans les fichiers Compose antérieurs à la version 3 (`cpu_shares`, `cpu_quota`, `cpuset`, `mem_limit`, `memswap_limit`, `mem_swappiness`).

Chacun de ceux-ci est une valeur unique, analogue à son homologue créer de service de docker.

Dans cet exemple général, le service redis est contraint de ne pas utiliser plus de 50 Mo de mémoire et 0,50 (50% d'un cœur) du temps de traitement disponible (CPU), et dispose de 20 Mo de mémoire et 0,25 heure de CPU réservée (comme toujours disponible). à cela).

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

Exceptions de mémoire insuffisante (OOME): Si less services ou les conteneurs tentent d'utiliser

plus de mémoire que le système n'en permet, on risque de rencontrer une exception de mémoire insuffisante et un conteneur, ou le démon Docker, peut être tué par le destructeur de OOM du noyau. Pour éviter que cela ne se produise, il faut s'assurer que l'application est exécutée sur des hôtes disposant d'une mémoire suffisante.

restart_policy

Configure pourquoi et comment redémarrer les conteneurs à leur sortie. Remplace le redémarrage.

- **condition**: l'une des valeurs suivantes: none, on-failure or any (par défaut: any).
- **delay**: combien de temps attendre entre les tentatives de redémarrage, spécifié en tant que durée (par défaut: 0).
- **max_attempts**: combien de fois tenter de redémarrer un conteneur avant d'abandonner (par défaut: ne jamais abandonner). Si le redémarrage échoue dans la fenêtre configurée, cette tentative ne compte pas dans la valeur configurée `max_attempts`. Par exemple, si `max_attempts` est défini sur “2” et que le redémarrage échoue à la première tentative, on peut tenter plus de deux redémarrages.
- **window**: combien de temps attendre avant de décider si un redémarrage a réussi, en tant que durée (par défaut: 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 et plus
```

Configure la manière dont le service devrait être annulé en cas d'échec de la mise à jour.

- **parallelism**: nombre de conteneurs à restaurer à la fois. Si la valeur est définie sur 0, tous les conteneurs sont restaurés simultanément.
- **delay**: délai d’attente entre les restaurations de chaque groupe de conteneurs (0 par défaut).
- **failure_action**: Que faire si une restauration échoue? `continue` ou `pause` (`pause` par défaut)
- **monitor**: durée après chaque mise à jour de la tâche pour surveiller l'échec (`ns | us | ms | s | m | h`) (0s par défaut).
- **max_failure_ratio**: taux d'échec à tolérer lors d'une restauration (valeur par défaut 0).
- **order**: Ordre des opérations lors des annulations. L'une des fonctions `stop-first` (l'ancienne tâche est arrêtée avant de commencer une nouvelle) ou `start-first` (la nouvelle tâche est démarrée en premier et les tâches en cours se chevauchent brièvement) (par défaut, `stop-first`).

update_config

Configure comment le service doit être mis à jour. Utile pour configurer les mises à jour roullback.

- **parallelism**: nombre de conteneurs à mettre à jour à la fois.
- **delay**: délai d'attente entre la mise à jour d'un groupe de conteneurs.
- **failure_action**: Que faire si une mise à jour échoue? Une des options `continue`, `restaure` ou `pause` (par défaut: `pause`).
- **monitor**: durée après chaque mise à jour de la tâche pour surveiller l'échec (`ns | us | ms | s | m | h`) (0s par défaut).
- **max_failure_ratio**: Taux d'échec à tolérer lors d'une mise à jour.
- **order**: Ordre d'opérations during mises à jour. L'une des fonctions `stop-first` (l'ancienne tâche est arrêtée avant d'en démarrer une nouvelle), ou `start-first` (la nouvelle tâche est démarrée en premier, et les tâches en cours se chevauchent brièvement) (valeur par défaut `stop-first`) pris en charge uniquement à partir de v3.4.

```
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
```

Options non prises en charge pour le déploiement de la pile de docker

Les sous-options suivantes (prises en charge pour `docker-compose` et `docker-compose run`) ne sont pas prises en charge pour le déploiement de la pile de docker ou la clé de déploiement.

```
    build
    cgroup_parent
    container_name
    devices
    tmpfs
    external_links
    links
    network_mode
    restart
    security_opt
    sysctls
    userns_mode
```

devices

Liste des mappages de périphériques. Utilise le même format que l'option de création du client docker `--device`.

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

cette option est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3).

depends_on

Dépendance explicite entre services, les dépendances de service provoquent les comportements suivants:

- `docker-compose up` démarre les services dans l'ordre des dépendances. Dans l'exemple suivant, `db` et `redis` sont démarrés avant `web`.
- `docker-compose up SERVICE` inclut automatiquement les dépendances de `SERVICE`. Dans l'exemple suivant, `docker-compose up web` crée et démarre également `db` et `redis`.
- `docker-compose stop` arre les services en ordre de dépendance. Dans l'exemple suivant, `web` est arrêté avant `db` et `redis`.

Exemple simple:

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

Il y a plusieurs choses à prendre en compte lors de l'utilisation de `depend_on`:

- `depend_on` n'attend pas que `db` et `redis` soient «prêtes» avant de démarrer `Web` - mais uniquement jusqu'à ce qu'elles aient été démarrées. Pour attendre qu'un service soit prêt, il faut contrôler l'ordre de démarrage.
- La version 3 ne prend plus en charge la forme de condition de `goes_on`.
- L'option `depend_on` est ignorée lors du déploiement d'une pile en mode swarm avec un fichier

Compose version 3.

dns

Serveurs DNS personnalisés. Peut être une valeur unique ou une liste.

`` dns: 8.8.8.8

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

dns_search

Domaines de recherche DNS personnalisés. Peut être une valeur unique ou une liste.

```
dns_search: exemple.com
```

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

entrypoint

Remplace le point d'entrée par défaut.

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

Le point d’entrée peut également être une liste, de la même manière que 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
```

La définition du point d'entrée remplace à la fois tout point d'entrée défini par défaut sur l'image du service par l'instruction ENTRYPOINT Dockerfile et efface toute commande par défaut de l'image - ce qui signifie que s'il existe une instruction CMD dans le Dockerfile, elle est ignorée.

env_file

Ajouter des variables d'environnement à partir d'un fichier. Peut être une valeur unique ou une liste.

Si on a spécifié un fichier Compose avec `docker-compose -f FILE`, les chemins dans `env_file` sont relatifs au répertoire dans lequel se trouve le fichier.

Les variables d'environnement déclarées dans la section d'environnement remplacent ces valeurs - cela est vrai même si ces valeurs sont vides ou non définies.

```
env_file: .env
```

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

Compose s'attend à ce que chaque ligne d'un fichier env soit au format VAR=VAL. Les lignes commençant par # sont traitées comme des commentaires et sont ignorées. Les lignes vides sont également ignorées.

```
# Définit l'environnement Rails/Rack
RACK_ENV=development
```

Si le service spécifie une option de construction, les variables définies dans les fichiers d'environnement ne sont pas automatiquement visibles lors de la construction. Utiliser la sous-option `args` de `build` pour définir les variables d'environnement de construction.

La valeur de VAL est utilisée telle quelle et n'est pas modifiée du tout. Par exemple, si la valeur est entourée de guillemets (comme c'est souvent le cas de variables shell), les guillemets sont inclus dans la valeur transmise à Compose.

L'ordre des fichiers dans la liste est important pour déterminer la valeur attribuée à une variable qui apparaît plusieurs fois. Les fichiers de la liste sont traités de haut en bas. Pour la même variable spécifiée dans le fichier `a.env` et affectée d'une valeur différente dans le fichier `b.env`, si `b.env` est répertorié après, la valeur de `b.env` est conservée. Par exemple, étant donné la déclaration suivante

dans docker-compose.yml:

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

Et les fichiers suivants:

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

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

$VAR contiendra hello.

environment

N'importe quel environnement peut être indiqué entre guillemets pour s'assurer qu'ils ne sont pas convertis en True ou False par l'analyseur YML.

Les variables d'environnement comportant uniquement une clé sont résolues à leurs valeurs sur l'ordinateur sur lequel Compose est en cours d'exécution, ce qui peut s'avérer utile pour les valeurs secrètes ou spécifiques à l'hôte.

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

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

Si le service spécifie une option de construction, les variables définies dans l'environnement ne sont pas automatiquement visibles lors de la construction.Utiliser la sous-option `args` de `build` pour définir les variables d'environnement de construction.

expose

Expose les ports sans les publier sur la machine hôte: ils ne seront accessibles qu'aux services liés et seul le port interne peut être spécifié.

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

external_links

Lien vers les conteneurs démarrés en dehors de ce fichier `docker-compose.yml` ou même en dehors de Compose, en particulier pour les conteneurs fournissant des services partagés ou communs.).

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

Lorsqu'on utilise le format de fichier version 2 ou supérieure, les conteneurs créés en externe doivent être connectés à au moins un des mêmes réseaux que le service qui les relie (les liens constituent une option héritée).

Cette option est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3).

Extra_hosts

Ajoute des mappages de nom d'hôte, en utilisant les mêmes valeurs que le paramètre client `--add-host` du menu fixe.

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

Une entrée avec l'adresse IP et le nom d'hôte est créée dans /etc/hosts à l'intérieur des conteneurs pour ce service, par exemple:

```
162.242.195.82  somehost
```

```
50.31.209.229   otherhost
```

healthcheck

```
version 2.1 et plus.
```

Configure une vérification exécutée pour déterminer si les conteneurs de ce service sont «sains».

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

- Les options `interval`, `timeout` et `start_period` sont spécifiés en tant que durées. (`start_period` est uniquement pris en charge par les versions v3.4 et supérieures du format de fichier compose).
- l'option `test` doit être une chaîne ou une liste. S'il s'agit d'une liste, le premier élément doit être `NONE`, `CMD` ou `CMD-SHELL`. Si c'est une chaîne, cela revient à spécifier `CMD-SHELL` suivi de cette chaîne.

```
# Test de l'application Web locale
Test: ["CMD", "curl", "-f", "http://localhost"]
```

Comme ci-dessus, mais enveloppé dans `/bin/sh` Les deux formes ci-dessous sont équivalentes.

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

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

Pour désactiver tout contrôle de l'intégrité par défaut défini par l'image, utiliser `disable: true`, ce qui revient à spécifier test: [“NONE”].

```
healthcheck:
  disable: true
```

image

Spécifie l'image à partir de laquelle le conteneur doit démarrer: il peut s'agir d'un repository/tag ou d'un ID d'image partiel.

```

```

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

Si l'image n'existe pas, Compose tente de l'extraire, à moins qu'on ait également spécifié le build, auquel cas elle la construit à l'aide des options spécifiées et la mrque avec le tag spécifié.

init

```
nouveau dans la version 3.7.
```

Exécute une init dans le conteneur qui transfère les signaux et les processus, puis définit cette option sur true pour activer cette fonctionnalité pour le service.

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

Le binaire init par défaut utilisé est `Tini` et est installé dans `/usr/libexec/docker-init` sur l'hôte du démon. On peut configurer le démon pour qu'il utilise un binaire init personnalisé via l'option de configuration `init-path`.

isolation

Spécifie la technologie d’isolation du conteneur. Sous Linux, la seule valeur prise en charge est la valeur par défaut. Sous Windows, les valeurs acceptées sont les valeurs par défaut, `processus` et `hyperv`.

labels

Ajoutez des métadonnées aux conteneurs à l'aide d'étiquettes Docker On peut utiliser un tableau ou un dictionnaire.

Il est recommandé d’utiliser la notation DNS inversée pour éviter que les étiquettes ne soient en conflit avec celles utilisées par d’autres logiciels.

```
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

Lien vers les conteneurs d'un autre service. Spécifier soit le nom du service et un alias de lien (`SERVICE: ALIAS`), soit simplement le nom du service.

Avertissement: L'indicateur `--link` est une fonctionnalité héritée de Docker.Il peut être éventuellement supprimé. À moins que l'on ne doive absolument continuer à l'utiliser, on recommande d'utiliser des réseaux définis par l'utilisateur pour faciliter la communication entre deux conteneurs au lieu d'utiliser `--link`. Une caractéristique que les réseaux définis par l'utilisateur ne prennent pas en charge mais que l'on peut utiliser avec `--link` est le partage de variables d'environnement entre les conteneurs. Cependant, on peut utiliser d'autres mécanismes tels que les volumes pour partager les variables d'environnement entre les conteneurs de manière plus contrôlée.

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

Les conteneurs du service liés sont accessibles à un nom d'hôte identique à l'alias ou au nom du service si aucun alias n'a été spécifié.

Les liens ne sont pas nécessaires pour permettre aux services de communiquer - par défaut, tout service peut atteindre tout autre service au nom de ce service.

Les liens expriment également la dépendance entre les services de la même manière que fait depend_on, ils déterminent donc l'ordre de démarrage du service.

Si ont définit les liens et les réseaux, les services ayant des liens entre eux doivent partager au moins un réseau en commun pour communiquer.

Cette option est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3).

logging

Configuration de la journalisation pour le service.

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

Le nom du pilote spécifie un pilote de journalisation pour les conteneurs du service, comme pour l'option `--log-driver` pour l'exécution du menu fixe.

La valeur par défaut est json-file.

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

Seuls les pilotes `json-file` et `journald` rendent les journaux disponibles directement à partir des journaux `docker-compose up` et `docker-compose`. L'utilisation d'un autre pilote n'imprime aucun journal.

Spécifier les options de journalisation du pilote de journalisation à l'aide de la clé options, comme pour l'option `--log-opt` pour l'exécution du menu fixe.

Les options de journalisation sont des paires clé-valeur. Un exemple d'options syslog:

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

Le fichier json du pilote par défaut contient des options permettant de limiter le nombre de journaux stockés. Pour ce faire, utiliser une paire clé-valeur pour la taille de stockage maximale et le nombre maximal de fichiers:

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

L'exemple ci-dessus stocke les fichiers journaux jusqu'à ce qu'ils atteignent une taille maximale de 200 Ko, puis les fait pivoter. La quantité de fichiers journaux individuels stockés est spécifiée par la valeur `max-file`. Lorsque les journaux dépassent les limites maximales, les anciens fichiers de journal sont supprimés pour permettre le stockage de nouveaux journaux.

Voici un exemple de fichier `docker-compose.yml` qui limite la journalisation du stockage:

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

les options de journalisation disponibles dépendent du pilote de journalisation utilisé. L'exemple ci-dessus de contrôle des fichiers journaux et des tailles utilise des options spécifiques au pilote de fichier json. Ces options particulières ne sont pas disponibles sur d'autres pilotes de journalisation.

network_mode

Mode réseau. Utilise les mêmes valeurs que le paramètre `--network` du client docker, plus le couple clé/valeur spécial `service:[service name]`.

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

Cette option est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3).

`mode_réseau: "host` ne peut pas être mélangé avec des liens.

networks

Réseaux à joindre, par référendement des entrées sous la clé `networks` de niveau supérieur.

```
services:
  some-service:
```

```
    networks:
     - some-network
     - other-network
```

alias

Alias (noms d'hôte alternatifs) pour ce service sur le réseau. Les autres conteneurs du même réseau peuvent utiliser le nom du service ou cet alias pour se connecter à l'un des conteneurs du service.

Etant donné que les alias sont étendus au réseau, un même service peut avoir différents alias sur différents réseaux.

un alias à l'échelle du réseau peut être partagé par plusieurs conteneurs, voire par plusieurs services. Si c'est le cas, le conteneur auquel le nom se résout n'est pas garanti.

Le format général est montré ici.

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

Dans l'exemple ci-dessous, trois services sont fournis (web, worker et db), ainsi que deux réseaux (new et legacy). Le service de base de données est accessible avec le nom d'hôte `db` ou `database` sur le réseau `new`, et avec le nom d'hôte `db` ou `mysql` sur le réseau `legacy`.

```
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

Spécifie une adresse IP statique pour les conteneurs de ce service lors de la connexion au réseau.

La configuration réseau correspondante dans la section réseaux de niveau supérieur doit comporter un bloc `ipam` avec des configurations de sous-réseau couvrant chaque adresse statique.

Si l'adressage IPv6 est utilisée, l'option `enable_ipv6` doit être définie et il faut utiliser un fichier Compose version 2.x. Les options IPv6 ne fonctionnent pas actuellement en mode swarm.

```
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"
```

Définit le mode PID sur le mode PID hôte. Cela active le partage entre le conteneur et le système d'exploitation hôte, l'espace d'adressage PID. Les conteneurs lancés avec cet indicateur peuvent accéder à d'autres conteneurs dans l'espace de noms de la machine et inversement, et en manipuler d'autres.

ports

Expose les ports.

Le mappage de port est incompatible avec `network_mode: host`

Syntaxe courte

Spécifier les deux ports (HOST: CONTAINER) ou simplement le port de conteneur (un port d'hôte éphémère est choisi).

lorsqu'on mappe des ports au format `HOST: CONTAINER`, on peut rencontrer des résultats erronés lorsqu'on utilise un port de conteneur inférieur à 60, car YAML analyse les nombres au format `xx: yy` en tant que valeur base-60. Pour cette raison, il est recommandé de toujours spécifier explicitement vos mappages de ports en tant que chaînes.

```
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"
```

Syntaxe longue

La syntaxe de forme longue permet la configuration de champs supplémentaires qui ne peuvent pas être exprimés sous forme courte.

- **cible**: le port à l'intérieur du conteneur
- **published**: le port exposé publiquement
- **protocole**: le protocole de port (tcp ou udp)

- **mode**: hôte pour la publication d'un port hôte sur chaque nœud, ou entrée pour un port en mode swarm à équilibrage de charge.

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

la syntaxe longue est nouvelle dans la v3.2

restart

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

- `no` est la stratégie de redémarrage par défaut et ne redémarre en aucun cas un conteneur.
- `always` le conteneur redémarre toujours.
- `on-failure` redémarre un conteneur si le code de sortie indique une erreur en cas d'échec.

cette option est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3). Utiliser restart_policy à la place.

secrets

Accorde l'accès aux secrets par service à l'aide de la configuration `per-service secrets`.

Le secret doit déjà exister ou être défini dans la configuration des secrets de niveau supérieur de ce fichier de pile, sinon le déploiement de la pile échoue.

Deux variantes de syntaxe différentes sont prises en charge.

Syntaxe courte

La variante de syntaxe courte spécifie uniquement le nom secret. Cela accorde au conteneur l'accès

au secret et le monte dans /run/secrets/<nom*du*scrit> dans le conteneur. Le nom source et le point de montage de destination sont tous deux définis sur le nom secret.

L'exemple suivant utilise la syntaxe abrégée pour accorder au service `redis` l'accès aux secrets `my_secret` et `my_other_secret`. La valeur de `my_secret` est définie sur le contenu du fichier `./my_secret.txt`, et `my_other_secret` est défini en tant que ressource externe, ce qui signifie qu'il a déjà été défini dans Docker, en exécutant la commande `docker secret create` ou par une autre pile.Si le secret externe n'existe pas, le déploiement de la pile échoue avec une erreur de secret non trouvé.

```
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
```

Syntaxe longue

La syntaxe longue fournit plus de précision dans la manière dont le secret est créé dans les conteneurs de tâches du service.

- **source**: nom du secret tel qu'il existe dans Docker.
- **cible**: nom du fichier à monter dans /run/secrets/ dans les conteneurs de tâches du service. La valeur par défaut est source si non spécifié.
- **uid et gid**: UID ou GID numérique qui possède le fichier dans `/run/secrets/` dans les conteneurs de tâches du service. Les deux valeurs par défaut sont 0 si elles ne sont pas spécifiées.
- **mode**: les autorisations pour le fichier à monter dans `/run/secrets/` dans les conteneurs de tâches du service, en notation octale. Par exemple, 0444 représente le monde lecture seule. La valeur par défaut dans Docker 1.13.1 est 0000, mais 0444 dans les versions les plus récentes. Les secrets ne peuvent pas être insérés en écriture car ils sont montés dans un système de fichiers temporaire. Par conséquent, lorsqu'on définit le bit en écriture, celui-ci est ignoré. Le bit exécutable peut être défini.

L'exemple suivant définit le nom du `my_secret` pour `redis_secret` dans le conteneur, définit le mode sur 0440 (lisible par un groupe) et définit l'utilisateur et le groupe sur 103. Le service redis n'a pas accès au secret my*other*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
```

On peut accorder à un service l'accès à plusieurs secrets et un peut combiner une syntaxe longue et une syntaxe courte. Définir un secret n'implique pas l'octroi d'un accès à un service.

security_opt

Remplace le schéma d'étiquetage par défaut pour chaque conteneur.

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

cette option est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3).

stop_grace_period

Indique combien de temps attendre lorsqu'on tente d'arrêter un conteneur s'il ne gère pas SIGTERM (ou le signal d'arrêt spécifié avec `stop_signal`) avant d'envoyer SIGKILL. Spécifié en tant que durée.

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

Par défaut, stop attend 10 secondes que le conteneur se ferme avant d'envoyer SIGKILL.

stop_signal

Définit un autre signal pour arrêter le conteneur. Par défaut, stop utilise `SIGTERM`. La définition d'un autre signal à l'aide de `stop_signal` fait que stop envoie ce signal à la place.

```
stop_signal: SIGUSR1
```

sysctls

Paramètres du noyau à définir dans le conteneur. On peut utiliser un tableau ou un dictionnaire.

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

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

cette option est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3).

tmpfs

version 2 et plus.

Monte un système de fichiers temporaire à l'intérieur du conteneur. Peut être une valeur unique ou une liste.

```
tmpfs: /run

tmpfs:
  - /run
  - /tmp
```

cette option est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3-3.5).

version 3.6 et plus

Monte un système de fichiers temporaire à l'intérieur du conteneur. Le paramètre `size` spécifie la taille en octets du montage tmpfs. Illimité par défaut.

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

ulimits

Remplace les `ulimits` (contrôles des ressources systèmes) par défaut pour un conteneur. On peut spécifier une limite unique en tant qu'entier ou des limites logicielles/matérielles en tant que mappage.

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

userns_mode

```
userns_mode: "host"
```

Désactive l'espace de nom d'utilisateur pour ce service si le démon Docker est configuré avec des espaces de nom d'utilisateur.

cette option est ignorée lors du déploiement d'une pile en mode swarm avec un fichier Compose (version 3).

volumes

Monte les chemins d’hôte ou les volumes nommés, spécifiés en tant que sous-options d’un service.

On peut monter un chemin d’hôte dans le cadre de la définition d’un service unique. Il n’est pas nécessaire de le définir dans la clé des volumes de niveau supérieur.

Toutefois, pour réutiliser un volume sur plusieurs services, il faut définir un volume nommé dans la clé

de volumes de niveau supérieur. Utiliser des volumes nommés avec des services, des swarms et des fichiers de pile.

la clé de volumes de niveau supérieur définit un volume nommé et le référence dans la liste des volumes de chaque service. Cela remplace `volumes_from` dans les versions antérieures du format de fichier Compose.

Cet exemple montre un volume nommé (mydata) utilisé par le service Web et un montage de liaison défini pour un seul service (premier chemin sous les volumes de service de base de données). Le service de base de données utilise également un volume nommé appelé dbdata (second chemin sous les volumes de service de base de données), mais le définit à l'aide de l'ancien format de chaîne pour le montage d'un volume nommé. Les volumes nommés doivent être répertoriés sous la clé de volumes de niveau supérieur, comme indiqué.

```
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:
```

Syntaxe courte

Spécifier éventuellement un chemin sur la machine hôte `HOST: CONTAINER` ou un mode d'accès `HOST: CONTAINER: ro`.

On peut monter un chemin relatif sur l'hôte, qui se développe par rapport au répertoire du fichier de configuration Compose utilisé. Les chemins relatifs doivent toujours commencer par `.` ou `..`.

```
volumes:
  # Spécifie simplement un chemin et laisse le moteur créer un volume
  - /var/lib/mysql

  # Spécifie un mappage de chemin absolu
  - /opt/data:/var/lib/mysql

  # Chemin sur l'hôte, par rapport au fichier Compose
  - ./cache:/tmp/cache

  # Chemin relatif à l'utilisateur
  - ~/configs:/etc/configs/:ro

  # Volume nommé
  - datavolume:/var/lib/mysql
```

Syntaxe longue

La forme longue syntax permet la configuration de champs supplémentaires qui ne peuvent pas être exprimés sous la forme abrégée.

- **type**: le type de montage volume, `bind` ou `tmpfs`
- **source**: la source du montage, un chemin sur l'hôte pour un montage de liaison ou le nom d'un volume défini dans la clé de volumes de niveau supérieur. Non applicable pour un montage tmpfs.
- **cible**: le chemin dans le conteneur où le volume est monté
- **read_only**: indicateur pour définir le volume en lecture seule
- **bind**: configurer des options de liaison supplémentaires
  - **propagation**: le mode de propagation utilisé pour la liaison
- **volume**: configurer des options de volume supplémentaires
  - **nocopy**: indicateur pour désactiver la copie des données d'un conteneur lors de la création d'un volume
- **tmpfs**: configure les options supplémentaires de tmpfs
  - **size**: taille en octets du montage tmpfs
- **consistency** les exigences de cohérence du montage, `consistent` (l'hôte et le conteneur ont une vue identique), `cached` (cache de lecture, la vue d'hôte fait autorité) ou `delegated` (cache de lecture-écriture, la vue du conteneur fait autorité)

```
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:
```

la syntaxe longue est nouvelle dans la v3.2

Volumes de services, swarms et fichiers de pile

Lorsqu'on avec des services, des essaims et des fichiers docker-stack.yml, les tâches (conteneurs) sauvegardant un service peuvent être déployées sur n'importe quel nœud d'un essaim, ce qui peut être différent à chaque mise à jour du service. .

En l'absence de nom des volumes avec les sources spécifiées, Docker crée un volume anonyme pour chaque tâche sauvegardant un service. Les volumes anonymes ne persistent pas après la suppression des conteneurs associés.

Pour que les données persistent, il faut utilisez=r un volume nommé et un pilote de volume prenant en charge plusieurs hôtes, afin que les données soient accessibles à partir de n'importe quel nœud. Vous pouvez également définir des contraintes sur le service afin que ses tâches soient déployées sur un nœud doté du volume.

Par exemple, le fichier docker-stack.yml pour l'exemple de voteapp dans Docker Labs définit un service appelé db qui exécute une base de données postgres. Il est configuré en tant que volume nommé pour conserver les données sur l'essaim et est contraint de ne s'exécuter que sur des nœuds de gestionnaire:

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

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

Chacune de ces options est une valeur unique, analogue à son homologue d'exécution du menu fixe. (`mac_address` est une option héritée).

```
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
```

Référence de configuration de volume

Bien qu'il soit possible de déclarer des volumes sur le fichier dans le cadre de la déclaration de service, cette section décrit comment créer des volumes nommés (sans recourir à `volumes_from`) pouvant être réutilisés sur plusieurs services, facilement récupérés et inspectés à l'aide de la ligne de commande ou de l'API docker.

Voici un exemple de configuration à deux services dans laquelle le répertoire de données d’une base de données est partagé avec un autre service en tant que volume afin de pouvoir être sauvegardé périodiquement:

```
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:
```

Une entrée sous la clé `volumes:` de niveau supérieur peut être vide. Dans ce cas, elle utilise le pilote par défaut configuré par le moteur (dans la plupart des cas, il s'agit du pilote local). On peut éventuellement le configurer avec les clés suivantes:

driver

Spécifie le pilote de volume à utiliser pour ce volume. Par défaut, quel que soit le pilote que le moteur Docker a été configuré pour utiliser, qui est dans la plupart des cas local. Si le pilote n'est pas disponible, le moteur renvoie une erreur lorsque docker-compose up tente de créer le volume.

```
driver: foobar
```

driver_opts

Spécifie une liste d'options sous forme de paires clé-valeur à transmettre au pilote pour ce volume. Ces options dépendent du pilote.

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

external

Si défini sur `true`, spécifie que ce volume a été créé en dehors de Compose. `docker-compose up` ne tente pas de le créer et génère une erreur s'il n'existe pas.

Pour les versions 3.3 et inférieures du format, external ne peut pas être utilisé avec d'autres clés de configuration de volume (`driver`, `driver_opts`, `labels`). Cette limitation n'existe plus pour les versions 3.4 et supérieures.

Dans l'exemple ci-dessous, au lieu de tenter de créer un volume appelé `[nom du projet]_data`, Compose recherche un volume existant simplement appelé `data` et le monte dans les conteneurs du service db.

```
version: "3.7"

services:
  db:
    image: postgres
```

```
    volumes:
      - data:/var/lib/postgresql/data

volumes:
  data:
    external: true
```

`external.name` est obsolète dans le format de fichier de la version 3.4, utiliser plutôt le nom.

On peut également spécifier le nom du volume séparément du nom utilisé pour le désigner dans le fichier Compose:

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

Les volumes externes sont toujours créés avec le déploiement de la pile de docker

Des volumes externes qui n'existent pas sont créés lorsqu'on utilise le déploiement de la pile de docker pour lancer l'application en mode swarm (au lieu de la composition du docker). En mode swarm, un volume est automatiquement créé lorsqu'il est défini par un service. Alors que les tâches de service sont planifiées sur de nouveaux nœuds, `swarmkit` crée le volume sur le nœud local.

labels

Ajouter des métadonnées aux conteneurs à l'aide d'étiquettes Docker. On peut utiliser un tableau ou un dictionnaire.

Il est recommandé d’utiliser la notation DNS inversée pour éviter que les étiquettes ne soient en conflit avec celles utilisées par d’autres logiciels.

```
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

```

```

```
Ajouté dans la version 3.4
```

Définit un nom personnalisé pour ce volume. Le champ de nom peut être utilisé pour référencer des volumes contenant des caractères spéciaux. Le nom est utilisé tel quel et ne sera pas défini avec le nom de la pile.

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

Il peut également être utilisé conjointement avec la propriété externe:

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

Référence de configuration du réseau

La clé `networks:` de niveau supérieur permet de spécifier les réseaux à créer.

driver

Spécifie le pilote à utiliser pour ce réseau.

Le pilote par défaut dépend de la configuration du moteur Docker que l'on utilise, mais dans la plupart des cas, il s'agit d'un pont (`bridge`) sur un hôte unique et `overlay` sur un Swarm.

Le moteur Docker renvoie une erreur si le pilote n'est pas disponible.

```
driver: overlay
```

bridge

Docker utilise par défaut un pont réseau sur un seul hôte.

overlay

Le pilote `overlay` crée un réseau nommé sur plusieurs nœuds d'un essaim.

host ou none

Utilise la pile de réseau de l'hôte ou aucun réseau. Équivalent à l'exécution de menu fixe `--net = host` ou `--net = none`. Utilisé uniquement lorsqu'on utilise des commandes de pile dans le menu fixe. Avec la commande `docker-compose`, utiliser plutôt `network_mode`.

Pour utiliser un réseau particulier sur une version commune, la syntaxe d'utilisation des réseaux intégrés, tels que `host` et `none`, est légèrement différente. Définir un réseau externe avec le nom `host` ou `none` (que Docker a déjà créé automatiquement) et un alias que Compose peut utiliser (`hostnet` ou `nonet` dans les exemples suivants), puis donner au service l'accès à ce réseau à l'aide de l'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

Spécifie une liste d'options sous forme de paires clé-valeur à transmettre au pilote de ce réseau. Ces options dépendent du pilote.

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

attachable

```
versions 3.2 et supérieures.
```

Utilisé uniquement lorsque le pilote est configuré pour se superposer. Si défini sur `true`, les conteneurs autonomes peuvent se connecter à ce réseau, en plus des services. Si un conteneur autonome est attaché à un réseau superposé, il peut communiquer avec des services et des conteneurs autonomes également attachés au réseau superposé à partir d'autres démons Docker.

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

enable_ipv6

```
Non pris en charge dans Compose File version 3
```

Activer la mise en réseau IPv6 sur ce réseau.

Pour utiliser `enable_ipv6`, vous devez utiliser un fichier Compose de version 2, car cette directive n'est pas encore prise en charge en mode Swarm.

ipam

Spécifie la configuration IPAM personnalisée. C'est un objet avec plusieurs propriétés, chacune d'elles étant optionnelle:

- **driver**: pilote IPAM personnalisé, au lieu du pilote par défaut.
- **config**: liste avec zéro ou plusieurs blocs de configuration, chacun contenant l'une des clés suivantes:
  - **subnet**: sous-réseau au format CIDR qui représente un segment de réseau

Un exemple complet:

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

Les configurations IPAM supplémentaires, telles que la passerelle, ne sont prises en compte que pour la version 2 pour le moment.

internal

Par défaut, Docker connecte également un réseau de pont à celui-ci pour fournir une connectivité externe. Pour créer un réseau superposé isolé de manière externe, définir cette option sur `true`.

labels

Ajoute des métadonnées aux conteneurs à l'aide d'étiquettes Docker. On peut utiliser un tableau ou un dictionnaire.

Il est recommandé d'utiliser la notation DNS inversée pour éviter que less étiquettes ne soient en conflit avec celles utilisées par d'autres logiciels.

```
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

Si défini sur `true`, spécifie que ce réseau a été créé en dehors de Compose. `docker-compose up` ne tente pas de le créer et génère une erreur s'il n'existe pas.

Pour les versions 3.3 et inférieures du format, `external` ne peut pas être utilisé avec d'autres clés de configuration réseau (`driver`, `driver_opts`, `ipam`, `internal`). Cette limitation n'existe plus pour les versions 3.4 et supérieures.

Dans l'exemple ci-dessous, le proxy est la passerelle vers le monde extérieur. Au lieu de tenter de créer un réseau appelé `[nom du projet]_outside`, Compose recherche un réseau existant simplement appelé `outside` et connecte les conteneurs du service `proxy` à celui-ci.

```
version: "3.7"

services:
```

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

networks:
  outside:
    external: true
```

external.name est déconseillé dans le format de fichier de la version 3.5, utiliser plutôt le nom.

On peut également spécifier le nom du réseau séparément du nom utilisé pour le désigner dans le fichier Compose:

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

name

```
Ajouté dans la version 3.5
```

Définit un nom personnalisé pour ce réseau. Le champ de nom peut être utilisé pour référencer des réseaux contenant des caractères spéciaux. Le nom est utilisé tel quel et ne sera pas défini avec le nom de la pile.

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

Il peut également être utilisé conjointement avec la propriété externe:

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

Références de configs

La déclaration `configs` de niveau supérieur définit ou fait référence aux configurations pouvant être attribuées aux services de cette pile. La source de la configuration est soit un fichier, soit externe.

- **file**: La configuration est créée avec le contenu du fichier dans le chemin spécifié.
- **external**: Si défini sur `true`, spécifie que cette configuration a déjà été créée. Docker n'essaie pas de le créer et s'il n'existe pas, une erreur de configuration non trouvée se produit.
- **name**: nom de l'objet de configuration dans Docker. Ce champ peut être utilisé pour référencer des configurations contenant des caractères spéciaux. Le nom est utilisé tel quel et ne sera pas défini avec le nom de la pile. Introduit dans le format de fichier version 3.5.

Dans cet exemple, `my_first_config` est créé (sous le nom `<stack_name>_my_first_config`) lorsque la pile est déployée et `my_second_config` existe déjà dans Docker.

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

Une autre variante pour les configurations externes est lorsque le nom de la configuration dans Docker est différent du nom existant dans le service. L'exemple suivant modifie le précédent pour utiliser la configuration externe appelée redis_config.

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

Il faut toujours accorder l'accès à la configuration à chaque service de la pile.

Références de secrets

La déclaration de secrets de niveau supérieur définit ou fait référence aux secrets pouvant être accordés aux services de cette pile. La source du secret est fichier ou externe.

- **file**: le secret est créé avec le contenu du fichier dans le chemin spécifié.
- **external**: Si défini sur `true`, spécifie que ce secret a déjà été créé. Docker n'essaie pas de le créer et s'il n'existe pas, une erreur secrète non trouvée se produit.
- **name**: nom de l'objet secret dans Docker. Ce champ peut être utilisé pour référencer des secrets contenant des caractères spéciaux. Le nom est utilisé tel quel et ne sera pas défini avec le nom de la pile. Introduit dans le format de fichier version 3.5.

Dans cet exemple, `my_first_secret` est créé sous le nom `<stack_name>_my_first_secret` lorsque la pile est déployée et `my_second_secret` existe déjà dans Docker.

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

Une autre variante pour les secrets externes est lorsque le nom du secret dans Docker est différent du nom existant dans le service. L'exemple suivant modifie le précédent pour utiliser le secret externe appelé `redis_secret`.

Compose File v3.5 et supérieur

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

Compose File v3.4 et inférieur

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

I lfaut toujours accorder l'accès aux secrets à chaque service de la pile.

Substitution des variables

Les options de configuration peuvent contenir des variables d’environnement. Compose utilise les valeurs de variable de l'environnement shell dans lequel docker-compose est exécuté. Par exemple, supposons que le shell contienne `POSTGRES_VERSION = 9.3` et qu'on fournisse cette configuration:

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

Lorsqu'on exécute `docker-compose up` avec cette configuration, Compose recherche la variable d’environnement `POSTGRES_VERSION` dans le shell et substitue sa valeur. Pour cet exemple, Compose résout l’image en `postgres: 9.3` avant d’exécuter la configuration.

Si aucune variable d'environnement n'est définie, Compose substitue par une chaîne vide. Dans l'exemple ci-dessus, si `POSTGRES_VERSION` n'est pas défini, la valeur de l'option image est `postgres :.`

On peut définir les valeurs par défaut des variables d'environnement à l'aide d'un fichier `.env`, que Compose recherche automatiquement. Les valeurs définies dans l'environnement shell remplacent celles définies dans le fichier `.env`.

La fonctionnalité de fichier `.env` ne fonctionne que lorsqu'on utilise la commande `docker-compose up` et ne fonctionne pas avec le déploiement de la pile de docker.

Les syntaxes `$VARIABLE` et `${VARIABLE}` sont supportées. De plus, lorsqu'on utilise le format de fichier 2.1, il est possible de fournir des valeurs par défaut en ligne en utilisant une syntaxe de shell typique:

- `${VARIABLE-default}` n'évalue la valeur par défaut que si `VARIABLE` n'est pas défini dans l'environnement.
- `${VARIABLE:-default}` est évalué à `default` si `VARIABLE` n'est pas défini **ou vide** dans l'environnement.

De même, la syntaxe suivante vous permet de spécifier des variables obligatoires:

- `${VARIABLE?Err}` se termine avec un message d'erreur contenant `err` si `VARIABLE` n'est pas défini dans l'environnement.
- `${VARIABLE:?Err}` se ferme avec un message d'erreur contenant `err` si `VARIABLE` est non défini **ou vide** dans l'environnement.

Les autres fonctionnalités de style shell étendues, telles que ${VARIABLE/foo/bar}, ne sont pas prises en charge.

Il faut utiliser un $$ (double signe dollar) quand la configuration nécessite un signe dollar littéral. Cela empêche également Compose d'interpoler une valeur. Par conséquent, une valeur $$ vous permet de faire référence à des variables d'environnement que l'on ne souhaite pas traiter par Compose.

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

Si on oublie et utilise un seul signe dollar ($), Compose interprète la valeur en tant que variable d'environnement et avertit:

```
The VAR_NOT_INTERPOLATED_BY_COMPOSE is not set. Substituting an empty
string.
```

Les Champs extent

```
Ajouté dans la version 3.4.
```

Il est possible de réutiliser des fragments de configuration à l'aide de champs `extent`. Ces champs spéciaux peuvent être de n'importe quel format, à condition qu'ils se trouvent à la racine du fichier Compose et que leur nom commence par la séquence de caractères x.

À partir des formats 3.7 (pour la série 3.x) et 2.4 (pour la série 2.x), les champs `extent` sont également autorisés à la racine des définitions de `service`, `volume`, `network`, `config` et `secret`.

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

Le contenu de ces champs est ignoré par Compose, mais ils peuvent être insérés dans les définitions de ressources à l'aide d'ancres YAML. Par exemple, pour que plusieurs services utilisent la même configuration de journalisation:

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

On peut écrire le fichier Compose comme suit:

```
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
```

Il est également possible de remplacer partiellement les valeurs dans les champs d'extension à l'aide du type YAML merge. Par exemple:

```
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
```