Use Docker to build Docker images all tiers

You can use GitLab CI/CD with Docker to create Docker images. For example, you can create a Docker image of your application, test it, and publish it to a container registry.

To run Docker commands in your CI/CD jobs, you must configure GitLab Runner to support `docker` commands.

If you want to build Docker images without enabling privileged mode on the runner, you can use a Docker alternative.

Enable Docker commands in your CI/CD jobs

To enable Docker commands for your CI/CD jobs, you can use:

- The shell executor
- Docker-in-Docker
- Docker socket binding

If you are using shared runners on GitLab.com, learn more about how these runners are configured.

Use the shell executor

To include Docker commands in your CI/CD jobs, you can configure your runner to use the `shell` executor. In this configuration, the `gitlab-runner` user runs the Docker commands, but needs permission to do so.

1. Install GitLab Runner.
2. Register a runner. Select the `shell` executor. For example:

```
sudo gitlab-runner register -n \
  --url https://gitlab.com/ \
  --registration-token REGISTRATION_TOKEN \
  --executor shell \
  --description "My Runner"
```

- On the server where GitLab Runner is installed, install Docker Engine. View a list of supported platforms.
- Add the `gitlab-runner` user to the `docker` group:

```
sudo usermod -aG docker gitlab-runner
```

Verify that `gitlab-runner` has access to Docker:

```
sudo -u gitlab-runner -H docker info
```

In GitLab, to verify that everything works, add `docker info` to `.gitlab-ci.yml`:

```
before_script:
  - docker info

build_image:
  script:
    - docker build -t my-docker-image .
    - docker run my-docker-image /script/to/run/tests
```

You can now use `docker` commands (and install `docker-compose` if needed).

When you add `gitlab-runner` to the `docker` group, you are effectively granting `gitlab-runner` full root permissions. Learn more about the security of the `docker` group.

Use Docker-in-Docker

"Docker-in-Docker" (`dind`) means:

- Your registered runner uses the Docker executor or the Kubernetes executor.
- The executor uses a container image of Docker, provided by Docker, to run your CI/CD jobs.

The Docker image has all of the `docker` tools installed and can run the job script in context of the image in privileged mode.

We recommend you use Docker-in-Docker with TLS enabled, which is supported by GitLab.com shared runners.

You should always specify a specific version of the image, like `docker:20.10.16`. If you use a tag like `docker:stable`, you have no control over which version is used. Unpredictable behavior can result, especially when new versions are released.

Use the Docker executor with Docker-in-Docker

You can use the Docker executor to run jobs in a Docker container.

Docker-in-Docker with TLS enabled in the Docker executor

Introduced in GitLab Runner 11.11.

The Docker daemon supports connections over TLS. In Docker 19.03.12 and later, TLS is the default.

This task enables `--docker-privileged`. When you do this, you are effectively disabling all of the security mechanisms of containers and exposing your host to privilege escalation. Doing this can lead to container breakout. For more information, see the official Docker documentation about runtime privilege and Linux capabilities.

To use Docker-in-Docker with TLS enabled:

1. Install GitLab Runner.
2. Register GitLab Runner from the command line. Use `docker` and `privileged` mode:

```
sudo gitlab-runner register -n \
  --url https://gitlab.com/ \
  --registration-token REGISTRATION_TOKEN \
  --executor docker \
  --description "My Docker Runner" \
  --docker-image "docker:20.10.16" \
  --docker-privileged \
  --docker-volumes "/certs/client"
```

- This command registers a new runner to use the `docker:20.10.16` image. To start the build and service containers, it uses the `privileged` mode. If you want to use Docker-in-Docker, you must always use `privileged = true` in your Docker containers.
- This command mounts `/certs/client` for the service and build container, which is needed for the Docker client to use the certificates in that directory. For more information on how Docker with TLS works, see https://hub.docker.com/_/docker/#tls .

The previous command creates a `config.toml` entry similar to this:

```
[[runners]]
  url = "https://gitlab.com/"
  token = TOKEN
  executor = "docker"
  [runners.docker]
    tls_verify = false
    image = "docker:20.10.16"
    privileged = true
    disable_cache = false
    volumes = ["/certs/client", "/cache"]
  [runners.cache]
    [runners.cache.s3]
    [runners.cache.gcs]
```

You can now use `docker` in the job script. Note the inclusion of the `docker:20.10.16-dind` service:

```
image: docker:20.10.16

variables:
  # When you use the dind service, you must instruct Docker to talk with
  # the daemon started inside of the service. The daemon is available
  # with a network connection instead of the default
  # /var/run/docker.sock socket. Docker 19.03 does this automatically
  # by setting the DOCKER_HOST in
  # https://github.com/docker-library/docker/blob
/d45051476babc297257df490d22cbd806f1b11e4/19.03/docker-entrypoint.
sh#L23-L29
  #
  # The 'docker' hostname is the alias of the service container as
described at
  # https://docs.gitlab.com/ee/ci/services/#accessing-the-services.
  #
  # Specify to Docker where to create the certificates. Docker
  # creates them automatically on boot, and creates
  # `/certs/client` to share between the service and job
  # container, thanks to volume mount from config.toml
  DOCKER_TLS_CERTDIR: "/certs"

services:
  - docker:20.10.16-dind

before_script:
```

```
  - docker info

build:
  stage: build
  script:
    - docker build -t my-docker-image .
    - docker run my-docker-image /script/to/run/tests
```

Docker-in-Docker with TLS disabled in the Docker executor

Sometimes you might have legitimate reasons to disable TLS. For example, you have no control over the GitLab Runner configuration that you are using.

Assuming that the runner's `config.toml` is similar to:

```
[[runners]]
  url = "https://gitlab.com/"
  token = TOKEN
  executor = "docker"
  [runners.docker]
    tls_verify = false
    image = "docker:20.10.16"
    privileged = true
    disable_cache = false
    volumes = ["/cache"]
  [runners.cache]
    [runners.cache.s3]
    [runners.cache.gcs]
```

You can now use `docker` in the job script. Note the inclusion of the `docker:20.10.16-dind` service:

```
image: docker:20.10.16

variables:
  # When using dind service, you must instruct docker to talk with the
  # daemon started inside of the service. The daemon is available with
  # a network connection instead of the default /var/run/docker.sock
socket.
  #
  # The 'docker' hostname is the alias of the service container as
described at
  # https://docs.gitlab.com/ee/ci/docker/using_docker_images.
html#accessing-the-services
  #
  # If you're using GitLab Runner 12.7 or earlier with the Kubernetes
executor and Kubernetes 1.6 or earlier,
  # the variable must be set to tcp://localhost:2375 because of how the
  # Kubernetes executor connects services to the job container
  # DOCKER_HOST: tcp://localhost:2375
```

```
  #
  DOCKER_HOST: tcp://docker:2375
  #
  # This instructs Docker not to start over TLS.
  DOCKER_TLS_CERTDIR: ""

services:
  - docker:20.10.16-dind

before_script:
  - docker info

build:
  stage: build
  script:
    - docker build -t my-docker-image .
    - docker run my-docker-image /script/to/run/tests
```

Use the Kubernetes executor with Docker-in-Docker

You can use the Kubernetes executor to run jobs in a Docker container.

Docker-in-Docker with TLS enabled in Kubernetes

Introduced in GitLab Runner Helm Chart 0.23.0.

To use Docker-in-Docker with TLS enabled in Kubernetes:

1. Using the Helm chart, update the `values.yml` file to specify a volume mount.

```
runners:
  config: |
    [[runners]]
      [runners.kubernetes]
        image = "ubuntu:20.04"
        privileged = true
      [[runners.kubernetes.volumes.empty_dir]]
        name = "docker-certs"
        mount_path = "/certs/client"
        medium = "Memory"
```

You can now use `docker` in the job script. Note the inclusion of the `docker:20.10.16-dind` service:

```
image: docker:20.10.16

variables:
  # When using dind service, you must instruct Docker to talk with
  # the daemon started inside of the service. The daemon is available
  # with a network connection instead of the default
  # /var/run/docker.sock socket.
```

```
  DOCKER_HOST: tcp://docker:2376
  #
  # The 'docker' hostname is the alias of the service container as
described at
  # https://docs.gitlab.com/ee/ci/services/#accessing-the-services.
  # If you're using GitLab Runner 12.7 or earlier with the Kubernetes
executor and Kubernetes 1.6 or earlier,
  # the variable must be set to tcp://localhost:2376 because of how the
  # Kubernetes executor connects services to the job container
  # DOCKER_HOST: tcp://localhost:2376
  #
  # Specify to Docker where to create the certificates. Docker
  # creates them automatically on boot, and creates
  # `/certs/client` to share between the service and job
  # container, thanks to volume mount from config.toml
  DOCKER_TLS_CERTDIR: "/certs"
  # These are usually specified by the entrypoint, however the
  # Kubernetes executor doesn't run entrypoints
  # https://gitlab.com/gitlab-org/gitlab-runner/-/issues/4125
  DOCKER_TLS_VERIFY: 1
  DOCKER_CERT_PATH: "$DOCKER_TLS_CERTDIR/client"

services:
  - docker:20.10.16-dind

before_script:
  - docker info

build:
  stage: build
  script:
    - docker build -t my-docker-image .
    - docker run my-docker-image /script/to/run/tests
```

Limitations of Docker-in-Docker

Docker-in-Docker is the recommended configuration, but is not without its own challenges:

- **The** `docker-compose` command: This command is not available in this configuration by default. To use `docker-compose` in your job scripts, follow the `docker-compose` installation instructions.
- **Cache**: Each job runs in a new environment. Concurrent jobs work fine, because every build gets its own instance of Docker engine and they don't conflict with each other. However, jobs can be slower because there's no caching of layers.
- **Storage drivers**: By default, earlier versions of Docker use the `vfs` storage driver, which copies the file system for each job. Docker 17.09 and later use `--storage-driver overlay2`, which is the recommended storage driver. See Using the OverlayFS driver for details.
- **Root file system**: Because the `docker:20.10.16-dind` container and the runner container don't share their root file system, you can use the job's working directory as a mount point for child containers. For example, if you have files you want to share with a child container, you might create a subdirectory under `/builds/$CI_PROJECT_PATH` and use it as your mount point. For a more detailed explanation, view issue #41227.

```
  variables:
    MOUNT_POINT: /builds/$CI_PROJECT_PATH/mnt
  script:
```

```
    - mkdir -p "$MOUNT_POINT"
    - docker run -v "$MOUNT_POINT:/mnt" my-docker-image
```

Use Docker socket binding

To use Docker commands in your CI/CD jobs, you can bind-mount `/var/run/docker.sock` into the container. Docker is then available in the context of the image.

If you bind the Docker socket and you are using GitLab Runner 11.11 or later, you can no longer use `docker:20.10.16-dind` as a service. Volume bindings are done to the services as well, making these incompatible.

Use the Docker executor with Docker socket binding

To make Docker available in the context of the image, you will need to mount `/var/run/docker.sock` into the launched containers. To do this with the Docker executor, you need to add `"/var/run/docker.sock:/var/run/docker.sock"` to the Volumes in the `[runners.docker]` section.

Your configuration should look something like this:

```
[[runners]]
  url = "https://gitlab.com/"
  token = RUNNER_TOKEN
  executor = "docker"
  [runners.docker]
    tls_verify = false
    image = "docker:20.10.16"
    privileged = false
    disable_cache = false
    volumes = ["/var/run/docker.sock:/var/run/docker.sock", "/cache"]
  [runners.cache]
    Insecure = false
```

You can also do this while registering your runner by providing the following options:

```
sudo gitlab-runner register -n \
  --url https://gitlab.com/ \
  --registration-token REGISTRATION_TOKEN \
  --executor docker \
  --description "My Docker Runner" \
  --docker-image "docker:20.10.16" \
  --docker-volumes /var/run/docker.sock:/var/run/docker.sock
```

Enable registry mirror for `docker:dind` service

When the Docker daemon starts inside of the service container, it uses the default configuration. You may want to configure a registry mirror for performance improvements and to ensure you don't reach Docker Hub rate limits.

The service in the `.gitlab-ci.yml` file

You can append extra CLI flags to the `dind` service to set the registry mirror:

```
```

```
services:
  - name: docker:20.10.16-dind
    command: ["--registry-mirror", "https://registry-mirror.example.
com"]  # Specify the registry mirror to use
```

The service in the GitLab Runner configuration file

Introduced in GitLab Runner 13.6.

If you are a GitLab Runner administrator, you can specify the `command` to configure the registry mirror for the Docker daemon. The `dind` service must be defined for the Docker or Kubernetes executor.

Docker:

```
[[runners]]
  ...
  executor = "docker"
  [runners.docker]
    ...
    privileged = true
    [[runners.docker.services]]
      name = "docker:20.10.16-dind"
      command = ["--registry-mirror", "https://registry-mirror.example.
com"]
```

Kubernetes:

```
[[runners]]
  ...
  name = "kubernetes"
  [runners.kubernetes]
    ...
    privileged = true
    [[runners.kubernetes.services]]
      name = "docker:20.10.16-dind"
      command = ["--registry-mirror", "https://registry-mirror.example.
com"]
```

The Docker executor in the GitLab Runner configuration file

If you are a GitLab Runner administrator, you can use the mirror for every `dind` service. Update the configuration to specify a volume mount.

For example, if you have a `/opt/docker/daemon.json` file with the following content:

```
{
  "registry-mirrors": [
    "https://registry-mirror.example.com"
```

```
  ]
}
```

Update the `config.toml` file to mount the file to `/etc/docker/daemon.json`. This would mount the file for **every** container that is created by GitLab Runner. The configuration is picked up by the `dind` service.

```
[[runners]]
  ...
  executor = "docker"
  [runners.docker]
    image = "alpine:3.12"
    privileged = true
    volumes = ["/opt/docker/daemon.json:/etc/docker/daemon.json:ro"]
```

The Kubernetes executor in the GitLab Runner configuration file

Introduced in GitLab Runner 13.6.

If you are a GitLab Runner administrator, you can use the mirror for every `dind` service. Update the configuration to specify a ConfigMap volume mount.

For example, if you have a `/tmp/daemon.json` file with the following content:

```
{
  "registry-mirrors": [
    "https://registry-mirror.example.com"
  ]
}
```

Create a ConfigMap with the content of this file. You can do this with a command like:

```
kubectl create configmap docker-daemon --namespace gitlab-runner --from-
file /tmp/daemon.json
```

Make sure to use the namespace that the Kubernetes executor for GitLab Runner uses to create job pods in.

After the ConfigMap is created, you can update the `config.toml` file to mount the file to `/etc/docker/daemon.json`. This update mounts the file for **every** container that is created by GitLab Runner. The configuration is picked up by the `dind` service.

```
[[runners]]
  ...
  executor = "kubernetes"
  [runners.kubernetes]
    image = "alpine:3.12"
    privileged = true
    [[runners.kubernetes.volumes.config_map]]
      name = "docker-daemon"
```

```
      mount_path = "/etc/docker/daemon.json"
      sub_path = "daemon.json"
```

Limitations of Docker socket binding

When you use Docker socket binding, you avoid running Docker in privileged mode. However, the implications of this method are:

- By sharing the Docker daemon, you are effectively disabling all the security mechanisms of containers and exposing your host to privilege escalation, which can lead to container breakout. For example, if a project ran `docker rm -f $(docker ps -a -q)` it would remove the GitLab Runner containers.
- Concurrent jobs may not work; if your tests create containers with specific names, they may conflict with each other.
- Any containers spawned by Docker commands are siblings of the runner rather than children of the runner. This may have complications and limitations that are unsuitable for your workflow.
- Sharing files and directories from the source repository into containers may not work as expected. Volume mounting is done in the context of the host machine, not the build container. For example:

```
 docker run --rm -t -i -v $(pwd)/src:/home/app/src test-image:latest
run_app_tests
```

You don't need to include the `docker:20.10.16-dind` service, like you do when you're using the Docker-in-Docker executor:

```
image: docker:20.10.16

before_script:
  - docker info

build:
  stage: build
  script:
    - docker build -t my-docker-image .
    - docker run my-docker-image /script/to/run/tests
```

Authenticate with registry in Docker-in-Docker

When you use Docker-in-Docker, the standard authentication methods don't work because a fresh Docker daemon is started with the service.

Option 1: Run `docker login`

In `before_script`, run `docker login`:

```
image: docker:20.10.16

variables:
  DOCKER_TLS_CERTDIR: "/certs"

services:
  - docker:20.10.16-dind

build:
```

```
  stage: build
  before_script:
    - echo "$DOCKER_REGISTRY_PASS" | docker login $DOCKER_REGISTRY --
username $DOCKER_REGISTRY_USER --password-stdin
  script:
    - docker build -t my-docker-image .
    - docker run my-docker-image /script/to/run/tests
```

To log in to Docker Hub, leave `$DOCKER_REGISTRY` empty or remove it.

Option 2: Mount `~/.docker/config.json` on each job

If you are an administrator for GitLab Runner, you can mount a file with the authentication configuration to `~/.docker/config.json`. Then every job that the runner picks up is authenticated already. If you are using the official `docker:20.10.16` image, the home directory is under `/root`.

If you mount the configuration file, any `docker` command that modifies the `~/.docker/config.json` fails. For example, `docker login` fails, because the file is mounted as read-only. Do not change it from read-only, because problems occur.

Here is an example of `/opt/.docker/config.json` that follows the `DOCKER_AUTH_CONFIG` documentation:

```
{
    "auths": {
        "https://index.docker.io/v1/": {
            "auth": "bXlfdXNlcm5hbWU6bXlfcGFzc3dvcmQ="
        }
    }
}
```

Docker

Update the volume mounts to include the file.

```
[[runners]]
  ...
  executor = "docker"
  [runners.docker]
    ...
    privileged = true
    volumes = ["/opt/.docker/config.json:/root/.docker/config.json:ro"]
```

Kubernetes

Create a ConfigMap with the content of this file. You can do this with a command like:

```
kubectl create configmap docker-client-config --namespace gitlab-runner
--from-file /opt/.docker/config.json
```

Update the volume mounts to include the file.

```
[[runners]]
  ...
  executor = "kubernetes"
  [runners.kubernetes]
    image = "alpine:3.12"
    privileged = true
    [[runners.kubernetes.volumes.config_map]]
      name = "docker-client-config"
      mount_path = "/root/.docker/config.json"
      # If you are running GitLab Runner 13.5
      # or lower you can remove this
      sub_path = "config.json"
```

Option 3: Use `DOCKER_AUTH_CONFIG`

If you already have `DOCKER_AUTH_CONFIG` defined, you can use the variable and save it in `~/.docker/config.json`.

There are multiple ways to define this authentication:

- In `pre_build_script` in the runner configuration file.
- In `before_script`.
- In `script`.

The following example shows `before_script`. The same commands apply for any solution you implement.

```
image: docker:20.10.16

variables:
  DOCKER_TLS_CERTDIR: "/certs"

services:
  - docker:20.10.16-dind

build:
  stage: build
  before_script:
    - mkdir -p $HOME/.docker
    - echo $DOCKER_AUTH_CONFIG > $HOME/.docker/config.json
  script:
    - docker build -t my-docker-image .
    - docker run my-docker-image /script/to/run/tests
```

Make Docker-in-Docker builds faster with Docker layer caching

When using Docker-in-Docker, Docker downloads all layers of your image every time you create a build. Recent versions of Docker (Docker 1.13 and later) can use a pre-existing image as a cache during the `docker build` step. This considerably speeds up the build process.

How Docker caching works

When running `docker build`, each command in `Dockerfile` results in a layer. These layers are kept around as a cache and can be reused if there haven't been any changes. Change in one layer causes all subsequent layers to be recreated.

You can specify a tagged image to be used as a cache source for the `docker build` command by using the `--cache-from` argument. Multiple images can be specified as a cache source by using multiple `--cache-from` arguments. Any image that's used with the `--cache-from` argument must first be pulled (using `docker pull`) before it can be used as a cache source.

Docker caching example

Here's a `.gitlab-ci.yml` file that shows how to use Docker caching:

```
image: docker:20.10.16

services:
  - docker:20.10.16-dind

variables:
  # Use TLS https://docs.gitlab.com/ee/ci/docker/using_docker_build.
html#tls-enabled
  DOCKER_HOST: tcp://docker:2376
  DOCKER_TLS_CERTDIR: "/certs"

before_script:
  - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
$CI_REGISTRY

build:
  stage: build
  script:
    - docker pull $CI_REGISTRY_IMAGE:latest || true
    - docker build --cache-from $CI_REGISTRY_IMAGE:latest --tag
$CI_REGISTRY_IMAGE:$CI_COMMIT_SHA --tag $CI_REGISTRY_IMAGE:latest .
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
    - docker push $CI_REGISTRY_IMAGE:latest
```

In the `script` section for the `build` stage:

1. The first command tries to pull the image from the registry so that it can be used as a cache for the `docker build` command.
2. The second command builds a Docker image by using the pulled image as a cache (see the `--cache-from $CI_REGISTRY_IMAGE:latest` argument) if available, and tags it.
3. The last two commands push the tagged Docker images to the container registry so that they may also be used as cache for subsequent builds.

Use the OverlayFS driver

The shared runners on GitLab.com use the `overlay2` driver by default.

By default, when using `docker:dind`, Docker uses the `vfs` storage driver which copies the file system on every run. This is a disk-intensive operation which can be avoided if a different driver is used, for example `overlay2`.

Requirements

1. Make sure a recent kernel is used, preferably `>= 4.2`.
2. Check whether the `overlay` module is loaded:

```
sudo lsmod | grep overlay
```

If you see no result, then it isn't loaded. To load it use:

```
sudo modprobe overlay
```

If everything went fine, you need to make sure module is loaded on reboot. On Ubuntu systems, this is done by editing `/etc/modules`. Just add the following line into it:

```
overlay
```

Use the OverlayFS driver per project

You can enable the driver for each project individually by using the `DOCKER_DRIVER` CI/CD variable in `.gitlab-ci.yml`:

```
variables:
  DOCKER_DRIVER: overlay2
```

Use the OverlayFS driver for every project

If you use your own runners, you can enable the driver for every project by setting the `DOCKER_DRIVER` environment variable in the `[[runners]]` section of the `config.toml` file:

```
environment = ["DOCKER_DRIVER=overlay2"]
```

If you're running multiple runners, you have to modify all configuration files.

Read more about the runner configuration and using the OverlayFS storage driver.

Docker alternatives

To build Docker images without enabling privileged mode on the runner, you can use one of these alternatives:

- `kaniko`.
- `buildah`.

For example, with `buildah`:

```
# Some details from https://major.io/2019/05/24/build-containers-in-
gitlab-ci-with-buildah/

build:
  stage: build
  image: quay.io/buildah/stable
```

```
  variables:
    # Use vfs with buildah. Docker offers overlayfs as a default, but
buildah
    # cannot stack overlayfs on top of another overlayfs filesystem.
    STORAGE_DRIVER: vfs
    # Write all image metadata in the docker format, not the standard
OCI format.
    # Newer versions of docker can handle the OCI format, but older
versions, like
    # the one shipped with Fedora 30, cannot handle the format.
    BUILDAH_FORMAT: docker
    # You may need this workaround for some errors:
https://stackoverflow.com/a/70438141/1233435
    BUILDAH_ISOLATION: chroot
    FQ_IMAGE_NAME: "${CI_REGISTRY_IMAGE}/test"
  before_script:
    # Log in to the GitLab container registry
    - export REGISTRY_AUTH_FILE=${HOME}/auth.json
    - echo "$CI_REGISTRY_PASSWORD" | buildah login -u
"$CI_REGISTRY_USER" --password-stdin $CI_REGISTRY
  script:
    - buildah images
    - buildah build -t $FQ_IMAGE_NAME
    - buildah images
    - buildah push $FQ_IMAGE_NAME
```

Use the GitLab Container Registry

After you've built a Docker image, you can push it up to the built-in GitLab Container Registry.

Troubleshooting

`docker: Cannot connect to the Docker daemon at tcp://docker:2375. Is the docker daemon running?`

This is a common error when you are using Docker-in-Docker v19.03 or later.

This issue occurs because Docker starts on TLS automatically.

- If this is your first time setting it up, read use the Docker executor with the Docker image.
- If you are upgrading from v18.09 or earlier, read our upgrade guide.

This error can also occur with the Kubernetes executor when attempts are made to access the Docker-in-Docker service before it has had time to fully start up. For a more detailed explanation, see this issue.

Docker `no such host` error

You may get an error that says `docker: error during connect: Post https://docker:2376/v1.40/containers/create: dial tcp: lookup docker on x.x.x.x:53: no such host`.

This issue can occur when the service's image name includes a registry hostname. For example:

```
image: docker:20.10.16
```

```
services:
  - registry.hub.docker.com/library/docker:20.10.16-dind
```

A service's hostname is derived from the full image name. However, the shorter service hostname `docker` is expected. To allow service resolution and access, add an explicit alias for the service name `docker`:

```
image: docker:20.10.16

services:
  - name: registry.hub.docker.com/library/docker:20.10.16-dind
    alias: docker
```