Store configuration data using Docker Configs

Estimated reading time: 18 minutes

About configs

Docker 17.06 introduces swarm service configs, which allow you to store non-sensitive information, such as configuration files, outside a service's image or running containers. This allows you to keep your images as generic as possible, without the need to bind-mount configuration files into the containers or use environment variables.

Configs operate in a similar way to secrets (https://docs.docker.com/engine/swarm/secrets/), except that they are not encrypted at rest and are mounted directly into the container's filesystem without the use of RAM disks. Configs can be added or removed from a service at any time, and services can share a config. You can even use configs in conjunction with environment variables or labels, for maximum flexibility.

> **Note**: Docker configs are only available to swarm services, not to standalone containers. To use this feature, consider adapting your container to run as a service with a scale of 1.

Configs are supported on both Linux and Windows services.

Windows support

Docker 17.06 and higher include support for configs on Windows containers. Where there are differences in the implementations, they are called out in the examples below. Keep the following notable differences in mind:

- Config files with custom targets are not directly bind-mounted into Windows containers, since Windows does not support non-directory file bind-mounts. Instead, configs for a container are all mounted in `C:\ProgramData\Docker\internal\configs` (an implementation detail which should not be relied upon by applications) within the container. Symbolic links are used to point from there to the desired target of the config within the container. The default target is `C:\ProgramData\Docker\configs`.
- When creating a service which uses Windows containers, the options to specify UID, GID, and mode are not supported for configs. Configs are currently only accessible by administrators and users with `system` access within the container.

How Docker manages configs

When you add a config to the swarm, Docker sends the config to the swarm manager over a mutual TLS connection. The config is stored in the Raft log, which is encrypted. The entire Raft log is replicated across the other managers, ensuring the same high availability guarantees for configs as for the rest of the swarm management data.

When you grant a newly-created or running service access to a config, the config is mounted as a file in the container. The location of the mount point within the container defaults to `/<config-name>` in Linux containers. In Windows containers, configs are all mounted into `C:\ProgramData\Docker\configs` and symbolic links are created to the desired location, which defaults to `C:\<config-name>`.

You can set the ownership (`uid` and `gid`) or the config, using either the numerical ID or the name of the user or group. You can also specify the file permissions (`mode`). These settings are ignored for Windows containers.

- If not set, the config is owned by the user and that running the container command (often `root`) and that user's default group (also often `root`).
- If not set, the config has world-readable permissions (mode `0444`), unless a `umask` is set within the container, in which case the mode is impacted by that `umask` value.

You can update a service to grant it access to additional configs or revoke its access to a given config at any time.

A node only has access to configs if the node is a swarm manager or if it is running service tasks which have been granted access to the config. When a container task stops running, the configs shared to it are unmounted from the in-memory filesystem for that container and flushed from the node's memory.

If a node loses connectivity to the swarm while it is running a task container with access to a config, the task container still has access to its configs, but cannot receive updates until the node reconnects to the swarm.

You can add or inspect an individual config at any time, or list all configs. You cannot remove a config that a running service is using. See Rotate a config (https://docs.docker.com/engine/swarm/configs/#example-rotate-a-config) for a way to remove a config without disrupting running services.

To update or roll back configs more easily, consider adding a version number or date to the config name. This is made easier by the ability to control the mount point of the config within a given container.

To update a stack, make changes to your Compose file, then re-run `docker stack deploy -c <new-compose-file> <stack-name>`. If you use a new config in that file, your services start using them. Keep in mind that configurations are immutable, so you can't change the file for an existing service. Instead, you create a new config to use a different file

You can run `docker stack rm` to stop the app and take down the stack. This removes any config that was created by `docker stack deploy` with the same stack name. This removes *all* configs, including those not referenced by services and those remaining after a `docker service update --config-rm`.

Read more about `docker config` commands

Use these links to read about specific commands, or continue to the example about using configs with a service (https://docs.docker.com/engine/swarm/configs/#example-use-configs-with-a-service).

- `docker config create` (https://docs.docker.com/engine/reference/commandline/config_create/)
- `docker config inspect` (https://docs.docker.com/engine/reference/commandline/config_inspect/)
- `docker config ls` (https://docs.docker.com/engine/reference/commandline/config_ls/)
- `docker config rm` (https://docs.docker.com/engine/reference/commandline/config_rm/)

Examples

This section includes graduated examples which illustrate how to use Docker configs.

> **Note**: These examples use a single-Engine swarm and unscaled services for simplicity. The examples use Linux containers, but Windows containers also support configs.

Defining and using configs in compose files

Both the `docker compose` and `docker stack` commands support defining configs in a compose file. See the Compose file reference (https://docs.docker.com/compose/compose-file/#configs) for details.

Simple example: Get started with configs

This simple example shows how configs work in just a few commands. For a real-world example, continue to Intermediate example: Use configs with a Nginx service (/engine/swarm/configs/#advanced-example-use-configs-with-a-nginx-service).

1. Add a config to Docker. The `docker config create` command reads standard input because the last argument, which represents the file to read the config from, is set to `-`.

   ```
   $ echo "This is a config" | docker config create my-config -
   ```

2. Create a `redis` service and grant it access to the config. By default, the container can access the config at `/my-config`, but you can customize the file name on the container using the `target` option.

   ```
   $ docker service create --name redis --config my-config redis:alpine
   ```

3. Verify that the task is running without issues using `docker service ps`. If everything is working, the output looks similar to this:

   ```
   $ docker service ps redis

   ID            NAME     IMAGE         NODE              DESIRED STATE  CURRENT STATE          ERROR  PORTS
   bkna6bpn8r1a  redis.1  redis:alpine  ip-172-31-46-109  Running        Running 8 seconds ago
   ```

4. Get the ID of the `redis` service task container using `docker ps`, so that you can use `docker container exec` to connect to the container and read the contents of the config data file, which defaults to being readable by all and has the same name as the name of the config. The first command below illustrates how to find the container ID, and the second and third commands use shell completion to do this automatically.

   ```
   $ docker ps --filter name=redis -q

   5cb1c2348a59

   $ docker container exec $(docker ps --filter name=redis -q) ls -l /my-config

   -r--r--r--    1 root     root            12 Jun  5 20:49 my-config

   $ docker container exec $(docker ps --filter name=redis -q) cat /my-config

   This is a config
   ```

5. Try removing the config. The removal fails because the `redis` service is running and has access to the config.

```
$ docker config ls

ID                          NAME                CREATED             UPDATED
fzwcfuqjkvo5foqu7ts7ls578   hello               31 minutes ago      31 minutes ago


$ docker config rm my-config

Error response from daemon: rpc error: code = 3 desc = config 'my-config' is
in use by the following service: redis
```

6. Remove access to the config from the running `redis` service by updating the service.

```
$ docker service update --config-rm my-config redis
```

7. Repeat steps 3 and 4 again, verifying that the service no longer has access to the config. The container ID is different, because the `service update` command redeploys the service.

```
$ docker container exec -it $(docker ps --filter name=redis -q) cat /my-config
cat: can't open '/my-config': No such file or directory
```

8. Stop and remove the service, and remove the config from Docker.

```
$ docker service rm redis

$ docker config rm my-config
```

Simple example: Use configs in a Windows service

This is a very simple example which shows how to use configs with a Microsoft IIS service running on Docker 17.06 EE on Microsoft Windows Server 2016 or Docker for Windows 17.06 CE on Microsoft Windows 10. It stores the webpage in a config.

This example assumes that you have PowerShell installed.

1. Save the following into a new file `index.html` .

```
<html>
  <head><title>Hello Docker</title></head>
  <body>
    <p>Hello Docker! You have deployed a HTML page.</p>
  </body>
</html>
```

2. If you have not already done so, initialize or join the swarm.

```
docker swarm init
```

3. Save the `index.html` file as a swarm config named `homepage` .

```
docker config create homepage index.html
```

4. Create an IIS service and grant it access to the `homepage` config.

```
docker service create
    --name my-iis
    --publish published=8000,target=8000
    --config src=homepage,target="\inetpub\wwwroot\index.html"
    microsoft/iis:nanoserver
```

5. Access the IIS service at `http://localhost:8000/` . It should serve the HTML content from the first step.

6. Remove the service and the config.

```
docker service rm my-iis

docker config rm homepage
```

Advanced example: Use configs with a Nginx service

This example is divided into two parts. The first part (/engine/swarm/configs/#generate-the-site-certificate) is all about generating the site certificate and does not directly involve Docker configs at all, but it sets up the second part (/engine/swarm/configs/#configure-the-nginx-container), where you store and use the site certificate as a series of secrets and the Nginx configuration as a config. The example shows how to set options on the config, such as the target location within the container and the file permissions (`mode`).

GENERATE THE SITE CERTIFICATE

Generate a root CA and TLS certificate and key for your site. For production sites, you may want to use a service such as `Let's Encrypt` to generate the TLS certificate and key, but this example uses command-line tools. This step is a little complicated, but is only a set-up step so that you have something to store as a Docker secret. If you want to skip these sub-steps, you can use Let's Encrypt (https://letsencrypt.org/getting-started/) to generate the site key and certificate, name the files `site.key` and `site.crt` , and skip to Configure the Nginx container (/engine/swarm/configs/#configure-the-nginx-container).

1. Generate a root key.

```
$ openssl genrsa -out "root-ca.key" 4096
```

2. Generate a CSR using the root key.

```
$ openssl req \
          -new -key "root-ca.key" \
          -out "root-ca.csr" -sha256 \
          -subj '/C=US/ST=CA/L=San Francisco/O=Docker/CN=Swarm Secret Example CA'
```

3. Configure the root CA. Edit a new file called `root-ca.cnf` and paste the following contents into it. This constrains the root CA to only sign leaf certificates and not intermediate CAs.

```
[root_ca]
basicConstraints = critical,CA:TRUE,pathlen:1
keyUsage = critical, nonRepudiation, cRLSign, keyCertSign
subjectKeyIdentifier=hash
```

4. Sign the certificate.

```
$ openssl x509 -req -days 3650 -in "root-ca.csr" \
               -signkey "root-ca.key" -sha256 -out "root-ca.crt" \
               -extfile "root-ca.cnf" -extensions \
               root_ca
```

5. Generate the site key.

```
$ openssl genrsa -out "site.key" 4096
```

6. Generate the site certificate and sign it with the site key.

```
$ openssl req -new -key "site.key" -out "site.csr" -sha256 \
          -subj '/C=US/ST=CA/L=San Francisco/O=Docker/CN=localhost'
```

7. Configure the site certificate. Edit a new file called `site.cnf` and paste the following contents into it. This constrains the site certificate so that it can only be used to authenticate a server and can't be used to sign certificates.

```
[server]
authorityKeyIdentifier=keyid,issuer
basicConstraints = critical,CA:FALSE
extendedKeyUsage=serverAuth
keyUsage = critical, digitalSignature, keyEncipherment
subjectAltName = DNS:localhost, IP:127.0.0.1
subjectKeyIdentifier=hash
```

8. Sign the site certificate.

```
$ openssl x509 -req -days 750 -in "site.csr" -sha256 \
    -CA "root-ca.crt" -CAkey "root-ca.key" -CAcreateserial \
    -out "site.crt" -extfile "site.cnf" -extensions server
```

9. The `site.csr` and `site.cnf` files are not needed by the Nginx service, but you need them if you want to generate a new site certificate. Protect the `root-ca.key` file.

CONFIGURE THE NGINX CONTAINER

1. Produce a very basic Nginx configuration that serves static files over HTTPS. The TLS certificate and key are stored as Docker secrets so that they can be rotated easily.

   In the current directory, create a new file called `site.conf` with the following contents:

```
server {
    listen               443 ssl;
    server_name          localhost;
    ssl_certificate      /run/secrets/site.crt;
    ssl_certificate_key  /run/secrets/site.key;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
}
```

2. Create two secrets, representing the key and the certificate. You can store any file as a secret as long as it is smaller than 500 KB. This allows you to decouple the key and certificate from the services that use them. In these examples, the secret name and the file name are the same.

```
$ docker secret create site.key site.key

$ docker secret create site.crt site.crt
```

3. Save the `site.conf` file in a Docker config. The first parameter is the name of the config, and the second parameter is the file to read it from.

```
$ docker config create site.conf site.conf
```

   List the configs:

```
$ docker config ls

ID                          NAME                CREATED             UPDATED
4ory233120ccg7biwvy11gl5z   site.conf           4 seconds ago       4 seconds ago
```

4. Create a service that runs Nginx and has access to the two secrets and the config. Set the mode to `0440` so that the file is only readable by its owner and that owner's group, not the world.

```
$ docker service create \
     --name nginx \
     --secret site.key \
     --secret site.crt \
     --config source=site.conf,target=/etc/nginx/conf.d/site.conf,mode=0440 \
     --publish published=3000,target=443 \
     nginx:latest \
     sh -c "exec nginx -g 'daemon off;'"
```

   Within the running containers, the following three files now exist:

   - `/run/secrets/site.key`
   - `/run/secrets/site.crt`
   - `/etc/nginx/conf.d/site.conf`

5. Verify that the Nginx service is running.

```
$ docker service ls

ID            NAME   MODE        REPLICAS  IMAGE
zeskcec62q24  nginx  replicated  1/1       nginx:latest

$ docker service ps nginx

NAME                  IMAGE         NODE  DESIRED STATE  CURRENT STATE          ERROR  PORTS
nginx.1.9ls3yo9ugcls  nginx:latest  moby  Running        Running 3 minutes ago
```

6. Verify that the service is operational: you can reach the Nginx server, and that the correct TLS certificate is being used.

```
$ curl --cacert root-ca.crt https://0.0.0.0:3000

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support, refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

```
$ openssl s_client -connect 0.0.0.0:3000 -CAfile root-ca.crt

CONNECTED(00000003)
depth=1 /C=US/ST=CA/L=San Francisco/O=Docker/CN=Swarm Secret Example CA
verify return:1
depth=0 /C=US/ST=CA/L=San Francisco/O=Docker/CN=localhost
verify return:1
---
Certificate chain
 0 s:/C=US/ST=CA/L=San Francisco/O=Docker/CN=localhost
   i:/C=US/ST=CA/L=San Francisco/O=Docker/CN=Swarm Secret Example CA
---
Server certificate
-----BEGIN CERTIFICATE-----
…
-----END CERTIFICATE-----
subject=/C=US/ST=CA/L=San Francisco/O=Docker/CN=localhost
issuer=/C=US/ST=CA/L=San Francisco/O=Docker/CN=Swarm Secret Example CA
---
No client certificate CA names sent
---
SSL handshake has read 1663 bytes and written 712 bytes
---
New, TLSv1/SSLv3, Cipher is AES256-SHA
Server public key is 4096 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1
    Cipher    : AES256-SHA
    Session-ID: A1A8BF35549C5715648A12FD7B7E3D861539316B03440187D9DA6C2E48822853
    Session-ID-ctx:
    Master-Key: F39D1B12274BA16D3A906F390A61438221E381952E9E1E05D3DD784F0135FB81353DA38C6D5C021CB926E844DFC49FC4
    Key-Arg   : None
    Start Time: 1481685096
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
```

7. Unless you are going to continue to the next example, clean up after running this example by removing the `nginx` service and the stored secrets and config.

```
$ docker service rm nginx

$ docker secret rm site.crt site.key

$ docker config rm site.conf
```

You have now configured a Nginx service with its configuration decoupled from its image. You could run multiple sites with exactly the same image but separate configurations, without the need to build a custom image at all.

Example: Rotate a config

To rotate a config, you first save a new config with a different name than the one that is currently in use. You then redeploy the service, removing the old config and adding the new config at the same mount point within the container. This example builds upon the previous one by rotating the `site.conf` configuration file.

1. Edit the `site.conf` file locally. Add `index.php` to the `index` line, and save the file.

```
server {
    listen               443 ssl;
    server_name          localhost;
    ssl_certificate      /run/secrets/site.crt;
    ssl_certificate_key  /run/secrets/site.key;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm index.php;
    }
}
```

2. Create a new Docker config using the new `site.conf`, called `site-v2.conf`.

```
$ docker config create site-v2.conf site.conf
```

3. Update the `nginx` service to use the new config instead of the old one.

```
$ docker service update \
  --config-rm site.conf \
  --config-add source=site-v2.conf,target=/etc/nginx/conf.d/site.conf,mode=0440 \
  nginx
```

4. Verify that the `nginx` service is fully re-deployed, using `docker service ls nginx`. When it is, you can remove the old `site.conf` config.

```
$ docker config rm site.conf
```

5. To clean up, you can remove the `nginx` service, as well as the secrets and configs.

```
$ docker service rm nginx

$ docker secret rm site.crt site.key

$ docker config rm site-v2.conf
```

You have now updated your `nginx` service's configuration without the need to rebuild its image.

swarm (https://docs.docker.com/glossary/?term=swarm), configuration (https://docs.docker.com/glossary/?term=configuration), configs (https://docs.docker.com/glossary/?term=configs)