2016-02-17

HANDLING PERMISSIONS WITH DOCKER VOLUMES

In this post I'll try to explain the method I use to avoid having permission issues when using Docker Volumes. This is pre Docker 1.10 (which added user namespaces) and I will talk about those in my next post.

Before we begin let me explain what are Docker Volumes and what they're used for. The official Docker docs explain this feature as follows:

```
A data volume is a specially-designated directory within one or more containers
that bypasses the Union File System.
```

The main use-case for volumes is for persisting data between container runs (seeing as container are ephemeral). This is useful for data directories when running databases (such as PostgreSQL) within containers. Other than persisting databases it's useful for sharing code folders from your host system to the container when running in your development environment.

However, there are 2 problems we have here:

1. If you write to the volume you won't be able to access the files that container has written because the process in the container usually runs as root.
2. You shouldn't run the process inside your containers as root but even if you run as

some hard-coded user it still won't match the user on your laptop/jenkins/staging.

The permissions problem is most annoying in development and testing environments because usually at some point you want to remove files that the process running in the container has created but you can't because on your laptop you're running as UID 1000 (on most Linux machines) and the files are owned either by UID 0 (root) or by some other UID that was perhaps hardcoded in the Dockerfile.

```
RUN  useradd --shell /bin/bash -u 1024 -o -c "" -m myuser
RUN mkdir -p /shared/tmp && chown user. /shared/ -R
USER myuser
CMD /usr/local/bin/myprocess
```

This solution is inadequate because you hard-code the UID of the user in the build process and even though your process won't be running as root it's still running as a user that's:

1. Not present on your local machine
2. The UID of the user is not 1000 (ie. your UID) and you still won't be able to cleanup files in the /shared/tmp

Docker provides a `-u` flag with it's `run` command to dynamically switch to a specified UID during container start. So we can write something like this:

```
deni@kanta:~$ docker run -it -u `id -u $USER` debian:jessie /bin/bash
I have no name!@dcb415bad433:/$ id
uid=1000 gid=0(root) groups=0(root)
```

This approach, while dynamic in the sense that the UID is specified at runtime, has 2 drawbacks:

1. The GID (group id) of the user is still 0 (root)
2. The UID 1000 is not present in the container's `/etc/passwd` file.

While no. 1. is definitely problematic for obvious reasons no. 2. is where we hit a wall. Now while the Linux Filesystem doesn't really care about user names, rather just UID's, some applications will refuse to start if the user is not present in `/etc/passwd`.

So what we need is something like `-u` but that doesn't just use the UID of our user but actually creates a user with that UID and then starts the process owned by it.

To do that we have to create a base Dockerfile from which all of our other Dockerfiles will inherit. That Dockerfile should look something like this.

```
FROM debian:jessie

RUN apt-get update && apt-get -y --no-install-recommends install \
    ca-certificates \
    curl

RUN gpg --keyserver ha.pool.sks-keyservers.net --recv-keys B42F6819007F00F88E364FD4036A9C25BF
357DD4
RUN curl -o /usr/local/bin/gosu -SL "https://github.com/tianon/gosu/releases/download/1.4/gos
u-$(dpkg --print-architecture)" \
    && curl -o /usr/local/bin/gosu.asc -SL "https://github.com/tianon/gosu/releases/download/
1.4/gosu-$(dpkg --print-architecture).asc" \
    && gpg --verify /usr/local/bin/gosu.asc \
    && rm /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu

COPY entrypoint.sh /usr/local/bin/entrypoint.sh

ENTRYPOINT ["/usr/local/bin/entrypoint.sh"]
```

In this base Dockerfile we're installing a tool called `gosu` and setting an entrypoint (https://docs.docker.com/engine/reference/builder/#entrypoint). An entrypoint is basically a script that gets executed before any other command that you might pass to your container. So unless we overwrite the entrypoint we are guaranteed to go through this script every time we launch our containers, before we actually run our actual process.

In fact the CMD statement from the Dockefile or from docker CLI gets passed to the `entrypoint.sh` script as command line arguments. The reason we're installing gosu is because we will need it to switch to the newly created user.

`NOTE`: The reason why we don't use sudo is explained in gosu repo's README.

Now let's look at the `entrypoint.sh` script:

```
#!/bin/bash

# Add local user
# Either use the LOCAL_USER_ID if passed in at runtime or
# fallback

USER_ID=${LOCAL_USER_ID:-9001}

echo "Starting with UID : $USER_ID"
useradd --shell /bin/bash -u $USER_ID -o -c "" -m user
export HOME=/home/user

exec /usr/local/bin/gosu user "$@"
```

What we're doing here is fetching a UID from an environment variable, defaulting to 9001 if it doesn't exist, and actually creating the user "user" with the familiar `useradd` command while setting it's UID explicitly.

And lastly we use `gosu` to execute our process `"$@"` as that user. Remember CMD from a Dockerfile or command from docker CLI gets passed to the entrypoint.sh script as command line arguments.

Now to build our base image:

```
deni@kanta:~$ docker build -t mybase .
```

And create our new child image:

```
FROM mybase

CMD ["/bin/bash"]
```

Build it:

```
deni@kanta:~$ docker build -t myimage .
```

Run it:

```
deni@kanta:~$ docker run -it myimage
Starting with UID : 9001
user@056b9bb45214:/$ id
uid=9001(user) gid=9001(user) groups=9001(user)
```

Run it with passing in our UID:

```
deni@kanta:~$ docker run -it -e LOCAL_USER_ID=`id -u $USER` myimage
Starting with UID : 1000
user@fc07b6c32b4f:/$ id
uid=1000(user) gid=1000(user) groups=1000(user)
```

Done! Now remember, the reason this works is because the Filesystem doesn't really care if the user is called "user" or "deni" or "jenkins". It only cares about the UID attached to that user, so the permissions will be preserved and various applications will not complain that there is no user with that UID.

Conclusion

When using docker containers it's a bad idea to run your processes as root (some applications even refuse to run as root). While running as root or any other hard-coded user it's hard to work with volume mounts because the files being written from within the container are going to be owned by a different user. That makes working with them or cleaning them up hard and needing to resort to sudo or similar. Which is increasingly annoying in development and CI environments.

In this post I've showed you a technique that you can use to build all of your images off of a base image (which you're probably already doing) that will allow you to start as whatever user you specify making sure to create that user in the process.

If a UID is specified, the container will start as that user, and if no UID is specified it will start as a default user with a random UID that should not collide with any existing users in docker images. (Aaand **it's over 9000!**)

So we're taking care of the permission issue and not allowing the containers to start as root all in one.

If this was helpful leave a comment down bellow and follow me on twitter (https://twitter.com/denibertovic).

docker (../../categories/docker/) root (../../categories/root/) sudo (../../categories/sudo/) volumes (../../categories/volumes/)

Mauro Rocco — Hi Denil, Nice hints, I added the link to this post to the celery community page http://www.celeryproject.org/c...

2 comments · 2 years ago

ogrimst — Thanks for the guide, now I have systemd on my server :)