# Docker Mastery with Kubernetes + Swarm (course in Udemy by Bret Fisher)

**File Permissions Across Multiple Containers**

At some point you'll have file permissions problems with container apps not having the permissions they need. Maybe you want multiple containers to access the same volume(s). Or maybe you're bind-mounting existing files into a container.

*Note that the below info is about pure Linux hosts, like production server setups. If you're using Docker Desktop locally, it will translate permissions from your host (macOS & Windows) into the container (Linux) automatically, but when working on pure Linux servers with just dockerd, no translation is made.*

How file permissions work across multiple containers accessing the same volume or bind-mount

**File ownership between containers and the host are just numbers.** They stay consistent no matter how you run them. Sometimes you see friendly user names in commands like ls but those are just name-to-number aliases that you'll see in `/etc/passwd` and `/etc/group`. Your host has those files, and usually, your containers will have their own. They are usually different. These files are really just for humans to see friendly names. The Linux Kernel only cares about IDs, which are attached to each file and directory in the file system itself, and those IDs are the same no matter which process accesses them.

When a container is just accessing its own files, this isn't usually an issue.

**But for multiple containers accessing the same volume or bind-mount, problems can arise in two ways:**

**1. Problem one: The `/etc/passwd` is different across containers.** Creating a named user in one container and running as that user may use ID 700, but that same name in another container with a different `/etc/passwd` may use a different ID for that same username. That's why I only care about IDs when trying to sync up permissions. You'll see this confusion if you're running a container on a Linux VM and it had a volume or bind-mount. If you do an ls on those files from the host, it may show them owned by ubuntu or node or systemd, etc. Then if you run ls inside the container, it may show a different friendly username. The IDs are the same in both cases, but the host will have a different passwd file than the container, and show you different friendly names. **Different names are fine, because it's only ID that counts. Two processes trying to access the same file must have a matching user ID or group ID.**

**2. Problem two: Your two containers are running as different users.** Maybe the user/group IDs and/or the USER statement in your Dockerfiles are different, and the two containers are technically running under different IDs. Different apps will end up running as different IDs. For example, the node base image creates a [user called node with ID of 1000](https://github.com/nodejs/docker-node/blob/6256caf2c507e7aafdeb8e7f837bab51f46f99e0/17/bullseye/Dockerfile#L3-L4), but the NGINX image creates an [nginx user as ID 101](https://github.com/nginxinc/docker-nginx/blob/6f0396c1e06837672698bc97865ffcea9dc841d5/mainline/debian/Dockerfile#L16-L17). Also, some apps spin-off sub-processes as different users. NGINX starts its main process (PID 1) as root (ID 0) but spawns sub-processes as the nginx user (ID 101), which keeps it more secure.

So for troubleshooting, this is what I do:

1. Use the command ps aux in each container to see a list of processes and usernames. The process needs a matching user ID or group ID to access the files in question.
2. Find the UID/GID in each containers `/etc/passwd` and `/etc/group` to translate names to numbers. You'll likely find there a miss-match, where one containers process originally wrote the files with its UID/GID and the other containers process is running as a different UID/GID.
3. Figure out a way to ensure both containers are running with either a matching user ID or group ID. This is often easier to manage in your own custom app (when using a language base image like python or node) rather than trying to change a 3rd party app's container (like nginx or postgres)... but it all depends. This may mean creating a new user in one Dockerfile and setting the startup user with USER. ([see USER docs](https://docs.docker.com/engine/reference/builder/#user)) The [node default image](https://github.com/nodejs/docker-node/blob/6256caf2c507e7aafdeb8e7f837bab51f46f99e0/17/bullseye/Dockerfile#L3-L4) has a good example of the commands for creating a user and group with hard-coded IDs:
4. RUN groupadd --gid 1000 node \\
5. && useradd --uid 1000 --gid node --shell /bin/bash --create-home node
6. USER 1000:1000

**Note**: When setting a Dockerfile's USER, use numbers, which work better in Kubernetes than using names.

**Note 2**: If ps doesn't work in your container, you may need to install it. In debian-based images with apt, you can add it with apt-get update && apt-get install procps