Docker development best practices

How to keep your images small

Small images are faster to pull over the network and faster to load into memory when starting containers or services. There are a few rules of thumb to keep image size small:

- Start with an appropriate base image. For instance, if you need a JDK, consider basing your image on the official `openjdk` image, rather than starting with a generic `ubuntu` image and installing `openjdk` as part of the Dockerfile.
- Use multistage builds. For instance, you can use the `maven` image to build your Java application, then reset to the `tomcat` image and copy the Java artifacts into the correct location to deploy your app, all in the same Dockerfile. This means that your final image doesn't include all of the libraries and dependencies pulled in by the build, but only the artifacts and the environment needed to run them.
    - If you need to use a version of Docker that does not include multistage builds, try to reduce the number of layers in your image by minimizing the number of separate `RUN` commands in your Dockerfile. You can do this by consolidating multiple commands into a single `RUN` line and using your shell's mechanisms to combine them together. Consider the following two fragments. The first creates two layers in the image, while the second only creates one.

      ```
      RUN apt-get -y update
      RUN apt-get install -y python
      ```

      ```
      RUN apt-get -y update && apt-get install -y python
      ```
- If you have multiple images with a lot in common, consider creating your own base image with the shared components, and basing your unique images on that. Docker only needs to load the common layers once, and they are cached. This means that your derivative images use memory on the Docker host more efficiently and load more quickly.
- To keep your production image lean but allow for debugging, consider using the production image as the base image for the debug image. Additional testing or debugging tooling can be added on top of the production image.
- When building images, always tag them with useful tags which codify version information, intended destination (`prod` or `test`, for instance), stability, or other information that is useful when deploying the application in different environments. Do not rely on the automatically-created `latest` tag.

Where and how to persist application data

- **Avoid** storing application data in your container's writable layer using storage drivers. This increases the size of your container and is less efficient from an I/O perspective than using volumes or bind

mounts.

- Instead, store data using volumes.
- One case where it is appropriate to use bind mounts is during development, when you may want to mount your source directory or a binary you just built into your container. For production, use a volume instead, mounting it into the same location as you mounted a bind mount during development.
- For production, use secrets to store sensitive application data used by services, and use configs for non-sensitive data such as configuration files. If you currently use standalone containers, consider migrating to use single-replica services, so that you can take advantage of these service-only features.

Use swarm services when possible

- When possible, design your application with the ability to scale using swarm services.
- Even if you only need to run a single instance of your application, swarm services provide several advantages over standalone containers. A service's configuration is declarative, and Docker is always working to keep the desired and actual state in sync.
- Networks and volumes can be connected and disconnected from swarm services, and Docker handles redeploying the individual service containers in a non-disruptive way. Standalone containers need to be manually stopped, removed, and recreated to accommodate configuration changes.
- Several features, such as the ability to store secrets and configs, are only available to services rather than standalone containers. These features allow you to keep your images as generic as possible and to avoid storing sensitive data within the Docker images or containers themselves.
- Let `docker stack deploy` handle any image pulls for you, instead of using `docker pull`. This way, your deployment doesn't try to pull from nodes that are down. Also, when new nodes are added to the swarm, images are pulled automatically.

There are limitations around sharing data amongst nodes of a swarm service. If you use Docker for AWS or Docker for Azure, you can use the Cloudstor plugin to share data amongst your swarm service nodes. You can also write your application data into a separate database which supports simultaneous updates.

Use CI/CD for testing and deployment

- When you check a change into source control or create a pull request, use Docker Cloud or another CI/CD pipeline to automatically build and tag a Docker image and test it. Docker Cloud can also deploy tested apps straight into production.

- Take this even further with Docker EE by requiring your development, testing, and security teams to sign images before they can be deployed into production. This way, you can be sure that before an image is deployed into production, it has been tested and signed off by, for instance, development, quality, and security teams.

Differences in development and production environments

Development	Production
Use bind mounts to give your container access to your source code.	Use volumes to store container data.

Development	Production
Use Docker for Mac or Docker for Windows.	Use Docker EE if possible, with userns mapping for greater isolation of Docker processes from host processes.
Don't worry about time drift.	Always run an NTP client on the Docker host and within each container process and sync them all to the same NTP server. If you use swarm services, also ensure that each Docker node syncs its clocks to the same time source as the containers.