Docker development best practices

The following development patterns have proven to be helpful for people building applications with Docker. If you have discovered something we should add, let us know.

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 CI/CD for testing and deployment

- When you check in a change to source control or create a pull request, use Docker Hub or another CI/CD pipeline to automatically build and tag a Docker image and test it.

- Take this even further with by requiring your development, testing, and security teams to sign images before they are deployed into production. This way, 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.
Use Docker Desktop for Mac or Docker Desktop for Windows.	Use Docker Engine, 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.