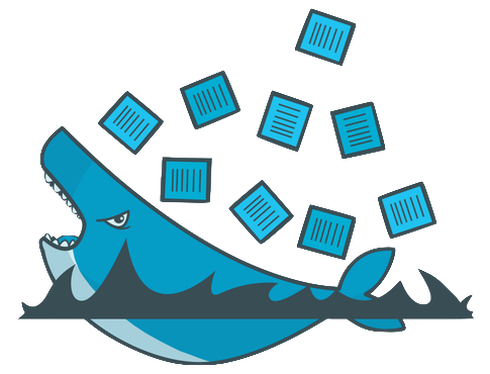
How NOT to use Docker

*March 29, 2018*



Docker as it stands, can be easily misunderstood due to ingrained traditional development practices. So, here lets explore how NOT to use Docker and the inherent principles that go along.

1. **Stateful Images** - Docker images should be stateless, the essence being if the contents of the mounted volume change, then the behavior of the container is altered. This makes every instance of the same containerized service to have different state leading to inconsistent behaviors.  
   As a result, *the service is portable, but not predictable*.

A corollary to this is the **traditional logging approach**, where the logs are dumped on the same container on which the service is deployed. And, the logs on the container are lost when the container is disposed off. In fact, *Containers are ephemeral i.e. short-lived, and hence disposable*. It doesn't matter how long they are up, rather you should expect them to go down at any moment and lose all data stored inside.

An alternate approach could be logging to data-volumes, but this would make it difficult to deploy these containers on different hosts, and also have complicated log aggregation issues due to distributed nature of deployments.

An optimum approach would be Dedicated Logging Containers which subscribe to containers' log events, aggregate them, then store (in an event-store) or forward the events to a third-party service.

1. **Inconstant / Mutable Images** - One of the key objectives of Docker is [immutable infrastructure](https://www.digitalocean.com/community/tutorials/what-is-immutable-infrastructure), for predictable and reliable infrastructure with consistent and repeatable deployment process.

An example could be - to update a service if an existing container is being upgraded it becomes a definite anti-pattern. To roll-out a newer version of the service, create new docker images and instances, and destroy the old instances.

Another anti-pattern would be tunneling to docker containers via ssh (using docker exec) to make changes to running containers such as modifying configurations or installing libraries. This may lead to undocumented and untraceable changes in the infrastructure without the means to re-play and automate them.

1. **Single Layer Image** - Docker provides for layered filesystem, which makes an image to be composed of multiple image-layers. For example, an image may have a base image layer for OS, another layer for the username definition, another layer for the runtime installation, another layer for the configuration, and a final layer for the application. It makes it easier to recreate, manage, and distribute docker images.

For a single layer image, the steps involved to create the image may not be known in entirety and hence may not be easy to replicate. As a corollary, never use docker commit command to create new images, as images created using this are non-reproducible. Instead make changes in the Dockerfile, terminate existing containers and start a new container with the updated image. *Never create Docker images in this fashion* - <https://www.techrepublic.com/article/how-to-commit-changes-to-a-docker-image/>.

1. **Bulky Images** - A large image will be harder to distribute, stress on having only the required files and libraries to run the containerized service. Don’t install unnecessary packages. Follow the mentioned [Dockerfile patterns / anti-patterns](https://gist.github.com/Faheetah/a2a401a01d2d56fa7d1a9d7ab0d2831b" \t "_blank), to refrain from adding unnecessary or avoidable layers.

1. **Latest Images** - The anti-pattern is relying on "latest" tagged images in either Dockerfile to create new images, or while running container images. Instead one should use tags which are encouraged because of the layered filesystem nature of containers. Using "tags" avoids surprises when a parent layer (in Dockerfile) is replaced by a new version which is not backward compatible, or when a wrong “latest” version is retrieved from the build cache. Another reason is, the deployed “latest” tag containers can’t be tracked on the version of the image it may be running.

1. **Images with Secrets** - hard-coding credentials in the image is not advisable, alternatively make use of environment variables to retrieve credentials from outside the container.

1. **Multi-Process Container** - Containers are units of encapsulation that do one job well. A container should run only one process so that the parts of the system can be independently scaled and updated. For running multiple processes together, you have virtual machines and not containers.

1. **Dependent Container Processes** - If the containerized services of an application system have a rigid start order, it again becomes an anti-pattern. For example, using a wait-for script in Dockerfile of a container service to ensure it waits to start until the database container is up. Such practice is discouraged because a containerized service should be resilient to external changes as the containers around may be terminated or started at any time.

A recommended approach is to make use of message brokers for inter-service communication calls. Also, identify highly-available (HA) components of the system and incorporate appropriate HA policies for them. Few such components could be service discovery, api gateway, message broker, configuration service, etc.

1. **Root Access Containers** - The host and the container share the same kernel. If the container running with root privileges is compromised, it can cause a big security hole. A suggested approach could be to create a specific user with limited privileges to run the containers, and towards the end of the Dockerfile script switch to this user (using USER instruction).

1. **Inconsistent Image Deployments** - Using different images, or even different tags in dev, test, staging and production environment may lead to missing single "source of truth". If the images being created are environment specific, that is different environments have different tagged / versioned images deployed, it may become hard to catch issues and complicate the provisioning process. A recommended approach would be to have a containerized configuration service, which would abstract the environment specific settings.

There is always a very close connection between how containers should operate and how a microservices application should behave, because both are built for the same purpose of providing flexibility, immutability, modularization and predictability of development and operations processes together. Microservices & Containers are the core to any successful DevOps Strategy. In a soon to be published blog(s), you can gauge into many such similarities - "Building Reactive Microservices".