**10 things to avoid in docker containers**

*Posted by*[*Rafael Benevides*](http://developers.redhat.com/blog/author/rafabene/)*on February 24, 2016*

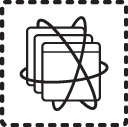
So you finally surrendered to containers and discovered that they solve a lot of problems and have a lot of **advantages**:

1. **First: Containers are immutable** – The OS, library versions, configurations, folders, and application are all wrapped inside the container. You guarantee that the same image that was tested in QA will reach the production environment with the same behaviour.
2. **Second: Containers are lightweight** – The memory footprint of a container is small. Instead of hundreds or thousands of MBs, the container will only allocate the memory for the main process.
3. **Third: Containers are fast** – You can start a container as fast as a typical linux process takes to start. Instead of minutes, you can start a new container in few seconds.

However, many users are still treating containers just like typical virtual machines and forget that containers have an important characteristic: **Containers are disposable**.

The *mantra* around containers:

***“Containers are ephemeral”.***



This characteristic forces users to change their mindset on how they should handle and manage containers; and I’ll explain what you should **NOT** do to keep extracting the best benefits of containers:

**1) Don’t store data in containers** – A container can be stopped, destroyed, or replaced. An application version 1.0 running in container should be easily replaced by the version 1.1 without any impact or loss of data. For that reason, if you need to store data, do it in a volume. In this case, you should also take care if two containers write data on the same volume as it could cause corruption.  Make sure your applications are designed to write to a shared data store.

**2) Don’t ship your application in two pieces** – As some people see containers like a virtual machine, most of them tend to think that they should deploy their application into existing running containers. That can be true during the *development* *phase* where you need to deploy and debug continuously; but for a continuous delivery (CD) pipeline to *QA* and p*roduction*, your application should be part of the image. Remember: Containers are **immutable**.

**3) Don’t create large images** – A large image will be harder to distribute. Make sure that you have only the required files and libraries to run your application/process. Don’t install unnecessary packages or run “updates”*(yum update)* that downloads many files to a new image layer.

**UPDATE:** There’s another post that better explain this recommendation: “[Keep it small: a closer look at Docker image sizing](http://developers.redhat.com/blog/2016/03/09/more-about-docker-images-size/)“

**4) Don’t use a single layer image** – To make effective use of the layered filesystem, always create your own base image layer for your OS, another layer for the [username definition](https://github.com/jboss-dockerfiles/base/blob/master/Dockerfile), another layer for the [runtime installation](https://github.com/jboss-dockerfiles/wildfly/blob/master/Dockerfile), another layer for the configuration, and finally another layer for your application. It will be easier to recreate, manage, and distribute your image.

**5) Don’t create images from running containers** – In other terms, don’t use “docker commit” to create an image. This method to create an image is not reproducible  and should be completely avoided. Always use a Dockerfile or any other S2I (source-to-image) approach that is totally reproducible, and you can track changes to the Dockerfile if you store it in a source control repository (git).

**6) Don’t use only the “latest” tag**– The latest tag is just like the “SNAPSHOT” for Maven users. Tags are encouraged because of the layered filesytem nature of containers. You don’t want to have surprises when you build your image some months later and figure out that your application can’t run because a parent layer (FROM in Dockerfile) was replaced by a new version that it’s not backward compatible or because a wrong “latest” version was retrieved from the build cache. The “latest” tag should also be avoided when deploying containers in production as you can’t track what version of the image is running.

**7) Don’t run more than one process in a single container** – Containers are perfect to run a single process (http daemon, application server, database), but if you have more than a single process, you may have more trouble managing, retrieving logs, and updating the processes individually.

**8) Don’t store credentials in the image. Use environment variables** – You don’t want to hardcode any username/password in your image. Use the environment variables to retrieve that information from outside the container. A great example of this principle is the [Postgres image](https://github.com/docker-library/postgres/blob/443c7947d548b1c607e06f7a75ca475de7ff3284/9.5/docker-entrypoint.sh).

**9) Don’t run processes as a root user** – *“By default docker containers run as root. (…) As docker matures, more secure default options may become available. For now, requiring root is dangerous for others and may not be available in all environments. Your image should use the USER instruction to specify a non-root user for containers to run as”*. (From [Guidance for Docker Image Authors](http://www.projectatomic.io/docs/docker-image-author-guidance/))

**10) Don’t rely on IP addresses** – Each container have their own internal IP address and it could change if you start and stop the container. If your application or microservice needs to communicate to another container, use environment variables to pass the proper hostname and port from one container to another.

For more information about containers, visit and register at [**http://developers.redhat.com/containers/**](http://developers.redhat.com/containers/)