Article
Luca Ravazzolo · Feb 2, 2018 4m read

Container - What is a Container IMAGE?

Container Images

In this second post on containers fundamentals, we take a look at what *container images* are.

What is a container image?

A container *image* is merely a binary representation of a container.

A running container or simply a *container* is the runtime state of the related container *image*.

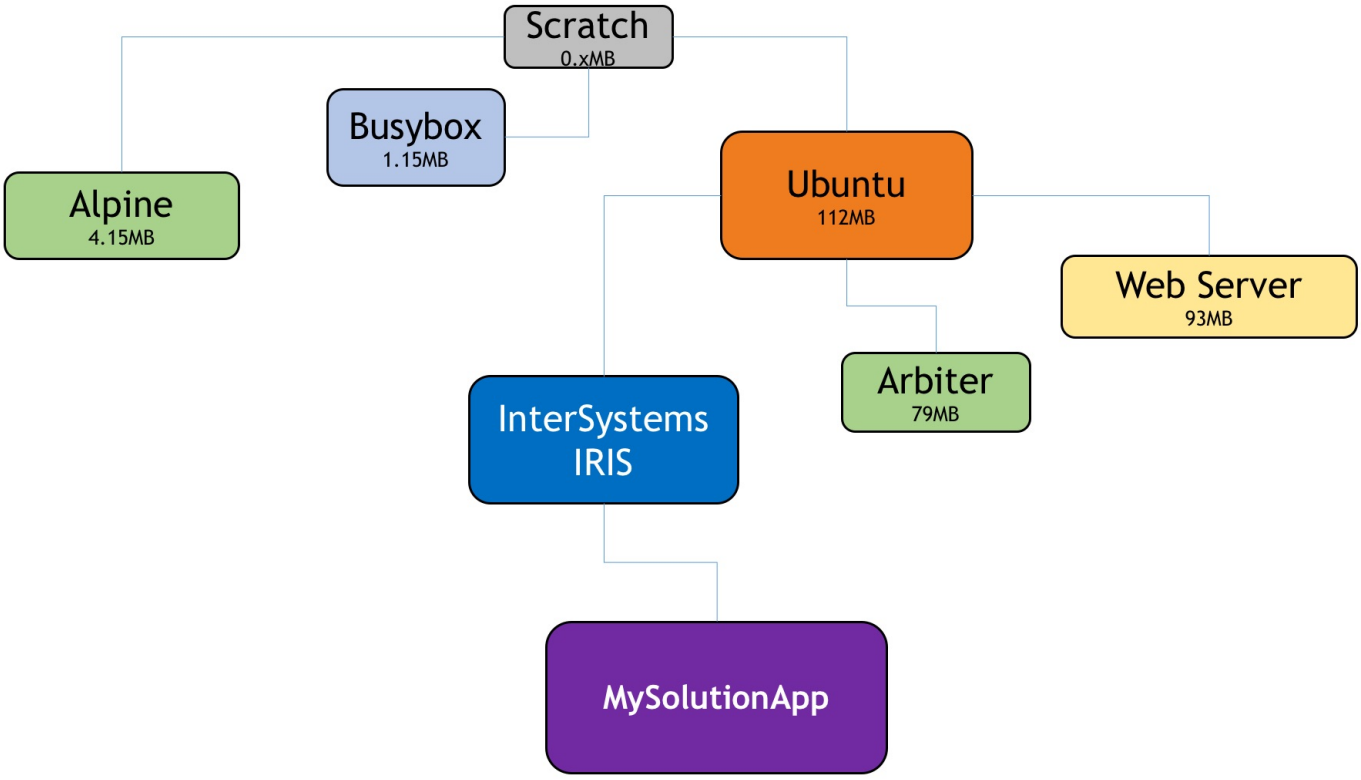
Please see the first post that explains what a container is.

Container images consist of a basic OS substratum and all the software we need to run our service. I use the term *service* for a given "container solution" as a generic description of a software solution wrapped in a container. I do that as I might be referring to a different type of "app-solutions" or components like a simple web server, a somewhat more complicated REST end-point or a fully blown monolith solution on its way to modernize the SDLC of a CI/CD provisioning pipeline. In a cloud-native, composable architecture, that might incorporate microservices; "service" then seems to be a more akin term.

We can think of a container image as that of a VMDK, OVA, OVF, QEMU, AWS AMI, etc. frozen package, however, container images are different with interesting characteristics.

With container images, we have the notion of a parent-child or derived image. Technically the notion is similar to that of snapshots: file blocks are added to an existing image. We end up with an image hierarchy that is very useful. Containers use Union-FS filesystem types that help us with the hierarchy and the depositing of extra layers.

There are clear advantages in organizing images in a tree structure (see the diagram below), we can all pick the fruit we like :) and of course one of the effects is that we can share them at each level.



Another way to think about container images and the image creation process is that it is very much like using an object-oriented language. It is the OO environment of sys-admins, or DevOps engineers I should say, as we inherit an image with all its properties and we further specify it. We also have the option to override the inherited components (think of an older library file with a vulnerability that needs to be fixed). If we picture that tree structure of inherited or derived images, we would know (and container tools can tell us) in which image there is an issue or a vulnerability. The problem can be quickly resolved at that particular image level, and all dependent images can then inherit the benefit of that fixed image upstream. Operationally the automated inheritance is achieved simply by a rebuilding process. The object-oriented analogy does not end here with images but extends to the running container. An image is like a class definition, while a container (see the previous post) is the instantiation -the object, of an image.

From the above, we derive that another advantage of the tree structure is that we can focus on each level or image for tailoring it.

Layers

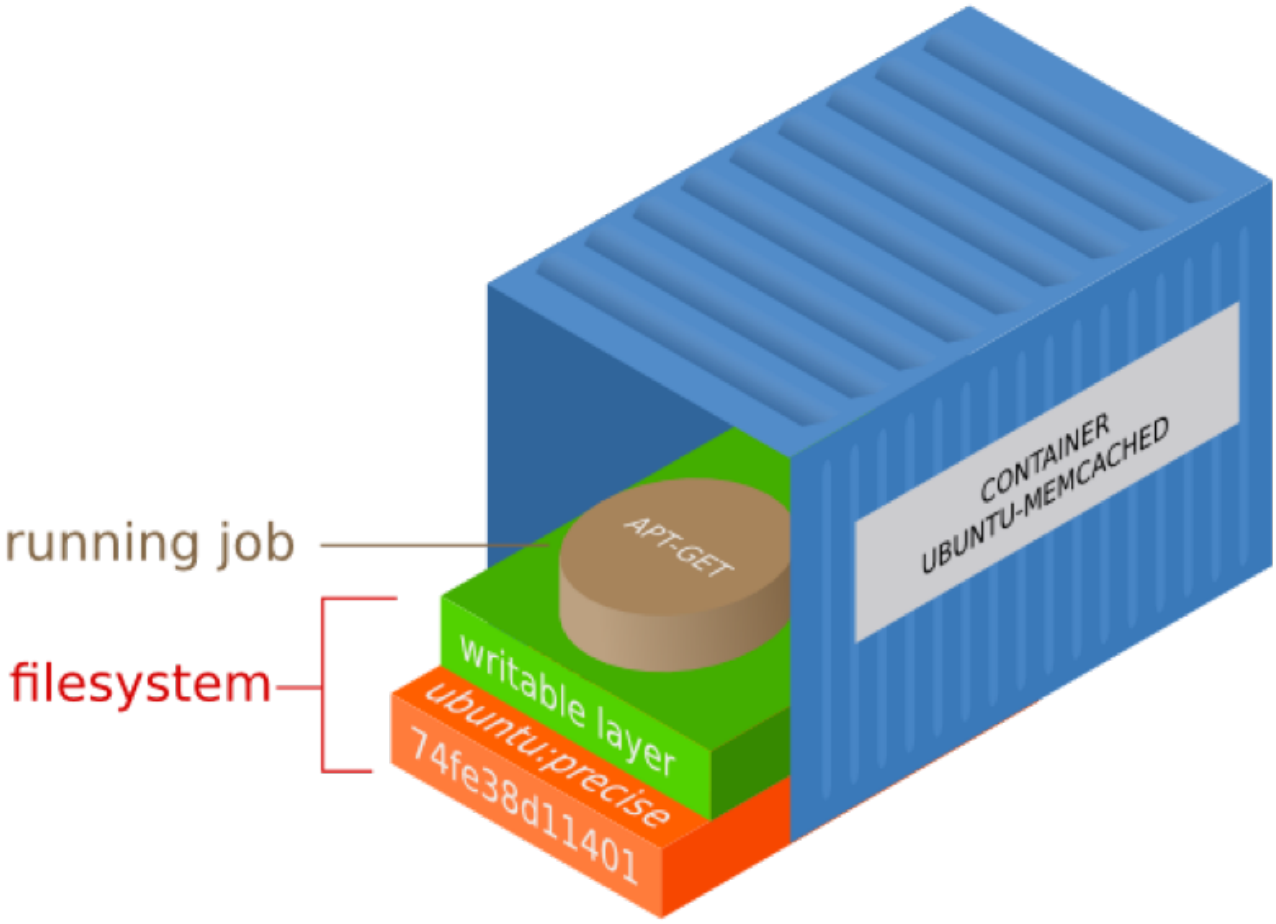
One other characteristic of a *container image* is that it is typically made of several *layers*.

What are layers? Layers are essentially files generated by running some commands.
You can view all the layers in the directory where Docker holds its artifacts which is under the Docker root directory, by default /var/lib/docker/.

The concept of layers is clever because they can be reused by multiple images and so it saves on storage requirements and it makes building new images & pulling them much faster because they are reused vs rebuilt or pulled again. This also improves the integrity of all your images.

Layers are also called *intemediate images*. When all intermediate images created by the build process are referenced together via the main name and tag nomenclature in use, we then have or are dealing with the *container image*.

Each container we run has a read-write top layer (see picture below) with a series of read-only layers underneath it. All those read-only layers are the result of commands executed when the container was built.



The reason why containers are called ephemeral is that usually, when we stop a container, we throw it away and that top read-write layer is lost forever. However, before removing the container you can commit your changes. The commit operation would save that top read-write layer into a read-only layer for the next run of the container. Effectively you now have a new container and are typically asked to save it with a new name & tag.

To summarize, Docker, the most widely used container engine & the company that has enabled us all to use containers easily, added a very intuitive abstraction to the low-level kernel APIs that also deals with the *packaging* of the solution. This is a special package (the *image*) as it is a *runnable* one. Docker offers a natural interaction with the Docker Engine (dockerd) via a simple REST API and an intuitive client CLI that allows us to deal with everything that relates to containers.

Docker helps us in.

- running them
    - $ docker run
- save, store & retrieve them
    - $ docker commit ¦ push ¦ pull
- create or build them
    - $ docker build

and add much more value to the whole "container experience".

In the next post, I'll cover the build process.
In the meantime, happy exploring of predefined images at Docker Hub.

#Best Practices #Cloud #Containerization #Deployment #DevOps #Docker #System Administration #InterSystems IRIS

Source URL:https://community.intersystems.com/post/container-what-container-image