Containers 101: Docker fundamentals

Wang, Chenxi . InfoWorld.com ; San Mateo (Jun 2, 2016).

ABSTRACT

[Download InfoWorld's essential guide to microservices and learn how to create modern Web and mobile applications that scale. | Cut to the key news in technology trends and IT breakthroughs with the InfoWorld Daily newsletter, our summary of the top tech happenings.] Using Docker, a developer can make a containerized application on his or her workstation, then easily deploy the container to any Docker-enabled server. Runtime changes, including any writes and updates to data and files, are saved in the container layer. [...]multiple concurrent running containers that share the same underlying image may have container layers that differ substantially. The Docker difference Docker's image format, extensive APIs for container management, and innovative software distribution mechanism have made it a popular platform for development and operations teams alike.

FULL TEXT

Docker started out in 2012 as an open source project, originally named dotcloud, to build single-application Linux containers. Since then, Docker has become an immensely popular development tool, increasingly used as a runtime environment. Few -- if any -- technologies have caught on with developers as quickly as Docker.

One reason Docker is so popular is that it delivers the promise of "develop once, run anywhere." Docker offers a simple way to package an application and its runtime dependencies into a single container; it also provides a runtime abstraction that enables the container to run across different versions of the Linux kernel.

[Download InfoWorld's essential guide to microservices and learn how to create modern Web and mobile applications that scale. | Cut to the key news in technology trends and IT breakthroughs with the InfoWorld Daily newsletter, our summary of the top tech happenings.]

Using Docker, a developer can make a containerized application on his or her workstation, then easily deploy the container to any Docker-enabled server. There's no need to retest or retune the container for the server environment, whether in the cloud or on premises.

In addition, Docker provides a software sharing and distribution mechanism that allows developers and operations teams to easily share and reuse container content. This distribution mechanism, coupled with portability across machines, helps account for Docker's popularity with operations teams and with developers.

Docker components

Docker is both a development tool and a runtime environment. To understand Docker, we must first understand the concept of a Docker container image. A container always starts with an image and is considered an instantiation of that image. An image is a static specification of what the container should be in runtime, including the application code inside the container and runtime configuration settings. Docker images contain read-only layers, which means that once an image is created it is never modified.

Figure 1 shows an example of a container image. This image depicts an Ubuntu image with an Apache installation. The image is a composition of three base Ubuntu layers plus an update layer, with an Apache layer and a custom file layer on top.

Figure 1: Docker image layers.

A running Docker container is an instantiation of an image. Containers derived from the same image are identical to each other in terms of their application code and runtime dependencies. But unlike images, which are read-only, running containers include a writable layer (the container layer) on top of the read-only content. Runtime changes, including any writes and updates to data and files, are saved in the container layer. Thus, multiple concurrent running containers that share the same underlying image may have container layers that differ substantially.
When a running container is deleted, the writable container layer is also deleted and will not persist. The only way to persist changes is to do an explicit docker commit command prior to deleting the container. When you do a docker commit, the running container content, including the writable layer, is written into a new container image and stored to the disk. This becomes a new image distinct from the image by which the container was instantiated.
Using this explicit docker commit command, one can create a successive, discrete set of Docker images, each built on top of the previous image. In addition, Docker uses a copy-on-write strategy to minimize the storage footprint of containers and images that share the same base components. This helps optimize storage space and minimize container start time.
Figure 2 depicts the difference between an image and a running container. Note that each running container can have a different writable layer.
Figure 2: Docker images and running Docker containers.
Beyond the image concept, Docker has a few specific components that are different from those in traditional Linux containers.
* **Docker daemon.** Also known as the Docker Engine, the Docker daemon is a thin layer between the containers and the Linux kernel. The Docker daemon is the persistent runtime environment that manages application containers. Any Docker container can run on any server that is Docker-daemon enabled, regardless of the underlying operating system.
* **Dockerfile.** Developers use Dockerfiles to build container images, which then become the basis of running containers. A Dockerfile is a text document that contains all of the configuration information and commands needed to assemble a container image. With a Dockerfile, the Docker daemon can automatically build a container image. This process greatly simplifies the steps for container creation.
More specifically, in a Dockerfile, you first specify a base image from which the build process starts. You then specify a succession of commands, after which a new container image can be built.
* **Docker command-line interface tools.** Docker provides a set of CLI commands for managing the lifecycle of image-based containers. Docker commands span development functions such as build, export, and tagging, as well as runtime functions such as running, deleting, starting and stopping a container, and more.
You can execute Docker commands against a particular Docker daemon or a registry. For instance, if you execute the docker -ps command, Docker will return a list of containers running on the daemon.
Content distribution with Docker
In addition to the runtime environment and container formats, Docker provides a software distribution mechanism, commonly known as a registry, that facilitates container content discovery and distribution.
The concept of registry is critical to the success of Docker, as it provides a set of utilities to pack, ship, store, discover, and reuse container content. Docker the company runs a public, free registry called the Docker Hub.
* **Registry.** A Docker registry is a place where container images are published and stored. A registry can be remote or on premises. It can be public, so everyone can use it, or private, restricted to an organization or a set of users. A Docker registry comes with a set of common APIs that allow users to build, publish, search, download, and manage container images.
* **Docker Hub.** Docker Hub is a public, cloud-based container registry managed by Docker. Docker Hub provides image discovery, distribution, and collaboration workflow support. In addition, Docker Hub has a set of official images that are certified by Docker. These are images from known software publishers such as Canonical, Red Hat, and MongoDB. You can use these official images as a basis for building your own images or applications.

Figure 3 depicts a workflow in which a user constructs an image and uploads it to the registry. Other users can pull the image from the registry to make production containers and deploy them to Docker hosts, wherever they are.
Figure 3: Docker registry workflow.
The immutability of Docker containers
One of the most interesting properties of Docker containers is their immutability and the resulting statelessness of containers.
As we described in the previous section, a Docker image, once created, does not change. A running container derived from the image has a writable layer that can temporarily store runtime changes. If the container is committed prior to deletion with docker commit, the changes in the writable layer will be saved into a new image that is distinct from the previous one.
Why is immutability good? Immutable images and containers lead to an immutable infrastructure, and an immutable infrastructure has many interesting benefits that are not achievable with traditional systems. These benefits include the following:
* **Version control.** By requiring explicit commits that generate new images, Docker forces you to do version control. You can keep track of successive versions of an image; rolling back to a previous image (therefore to a previous system component) is entirely possible, as previous images are kept and never modified.
* **Cleaner updates and more manageable state changes.** With immutable infrastructure, you no longer have to upgrade your server infrastructure, which means no need to change configuration files, no software updates, no operating system upgrades, and so on. When changes are needed, you simply make new containers and push them out to replace the old ones. This is a much more discrete and manageable method for state change.
* **Minimized drift.** To avoid drift, you can periodically and proactively refresh all the components in your system to ensure they are the latest versions. This practice is a lot easier with containers that encapsulate smaller components of the system than it is with traditional, bulky software.
The Docker difference
Docker's image format, extensive APIs for container management, and innovative software distribution mechanism have made it a popular platform for development and operations teams alike. Docker brings these notable benefits to an organization.
* **Minimal, declarative systems.** Docker containers are at their best if they are small, single-purpose applications. This gives rise to containers that are minimal in size, which in turn support rapid delivery, continuous integration, and continuous deployment.
* **Predictable operations.** The biggest headache of system operations has always been the seemingly random behavior of infrastructure or applications. By forcing you to make smaller, more manageable updates and by providing a mechanism to minimize system drift, Docker helps you build more predictable systems. When drifts are eliminated, you have assurance that the software will always behave in an identical manner, no matter how many times you deploy it.
* **Extensive software reuse.** Docker containers reuse layers from other images, which naturally promotes software reuse. The sharing of Docker images via registries is another great example of component reuse, on a grand scale.
* **True multicloud portability.** Docker enables true platform independence, by allowing containers to migrate freely between different cloud platforms, on-premises infrastructures, and development workstations.
Docker is already changing the way organizations build systems and deliver services. It's starting to reshape the way we think about software design and the economics of software delivery. Before these changes truly take root, organizations need to better understand how to manage security and policies for the Docker environment. But that's a topic for another article.
Chenxi Wang is chief strategy officer for container security firm Twistlock.
New Tech Forum provides a venue to explore and discuss emerging enterprise technology in unprecedented depth and breadth. The selection is subjective, based on our pick of the technologies we believe to be important and of greatest interest to InfoWorld readers. InfoWorld does not accept marketing collateral for publication and reserves

the right to edit all contributed content. Send all inquiries to newtechforum@infoworld.com.
Credit: By Chenxi Wang

DETAILS

Subject:	Software upgrading; Infrastructure; Linux
Company / organization:	Name: InfoWorld; NAICS: 511120
Publication title:	InfoWorld.com; San Mateo
Publication year:	2016
Publication date:	Jun 2, 2016
Publisher:	Infoworld Media Group
Place of publication:	San Mateo
Country of publication:	United States, San Mateo
Publication subject:	Computers--Microcomputers, Computers--Computer Industry
Source type:	Trade Journals
Language of publication:	English
Document type:	News
ProQuest document ID:	1793553179
Document URL:	https://login.ezproxy.napier.ac.uk/login?url=https://search.proquest.com/docview/1793553179?accountid=16607
Copyright:	Copyright Infoworld Media Group Jun 2, 2016
Last updated:	2016-06-03
Database:	ABI/INFORM Collection,SciTech Premium Collection

LINKS

Check LibrarySearch for availability