Global Knowledge®

Expert Reference Series of White Papers

Software Problems and How Docker Addresses Them

Software Problems and How Docker Addresses Them

Jon Gallagher, Global Knowledge Instructor, Certified AWS Solutions Architect, Certified AWS SysOps Administrator, Authorized Amazon Instructor

Introduction

Docker is a new approach to old, but increasingly troublesome problems in the software industry, namely:

- How can we deploy ever more powerful and complex software systems that are used by tens, hundreds, or thousands of users concurrently?
- How can we create, update, and maintain this software, while giving developers the platforms to run the software for testing and debugging?
- How can we facilitate testing and create automated systems that detect bugs and performance problems?
- "How can we deploy these systems, doing the system administration equivalent of changing tires on a moving car, to help users who depend on our software to always be available?
- How can we use the lessons we learned in creating more powerful and flexible hardware to help us solve our software problems?

That last question is an important one, because hardware went through a similar evolution to address similar issue and, Docker was partly inspired by the hardware evolution. As hardware became more powerful, the IT industry used that extra power to solve problems inherent in running complex systems.

The hardware was "chopped up" into virtual machine (VM) software that for all intents and purposes is a separate machine, indistinguishable from those running on traditional machines (bare metal computers). VMs solved the problem of making computers more efficient and cost-effective. You can buy one or more large boxes, then divide their capacity into multiple smaller VMs to run the system. As the system grows and changes, just change the space and power allocated to the VM.

To be as portable as possible a VM defines the operating system to be used, the number of CPUs that need to be allocated, the amount of memory that must be assigned, and any local storage space reserved for it. When these resources become available the VM boots up the operating system, starts any other necessary programs, then is ready to run anything a bare metal computer might run.

On the software side, new systems are also becoming more complex and as such they are increasingly depending on other software systems (for example, an application that depends on an image-rendering library). These dependencies must be managed carefully, because a misconfigured system may not run, may run incorrectly, or may have security vulnerabilities. One way to manage dependencies is to use the VM approach: just package up the desired software, along with all the software it depends on, into a VM image. Then when the system boots up, everything is in the correct place at the correct version level with the correct configuration.

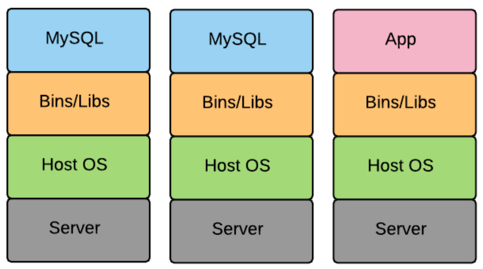
The trouble with the VM approach for deploying software is that a server must be built with each package. The packages end up getting bigger and more complex to justify the resources dedicated to starting and running the VM. Also, deploying software as a VM means that all the resources for the VM itself must be allocated, making it difficult to have multiple VMs on a developer's laptop, for example.

Meanwhile, with the concept of minimum viable products (MVPs) and agile approaches to development, project stakeholders and end users are demanding faster cycle times and more responsive software teams. Rather than waiting months or years for products, users are demanding product releases in weeks, days, and maybe even hours or minutes. Think of how often companies like Google, Twitter, Amazon, and Facebook change their software. In these companies, there is no concept of a release being frozen, tested, and then released. The software is continuously changing. And because there is no "frozen" version of the software, each new iteration of the software must be packaged so it can be deployed quickly.

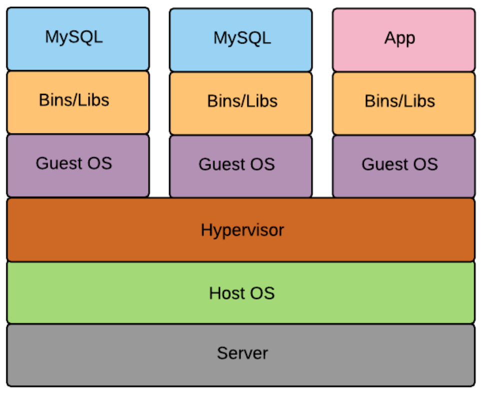
At the same time, the idea that a company would develop a single software system based on one programming language, one set of libraries, for one operating system, is no longer standard practice. Now, each group working on its own modules within a software system can choose the tools and environments that best meet the modules' needs. This new paradigm means that all the dependencies for any software, such as libraries, run-time environments as well as the new code itself, must be part of the release.

Finally, developers, testers, and administrators must be able to create running versions of these software systems to develop, test, and run in production.

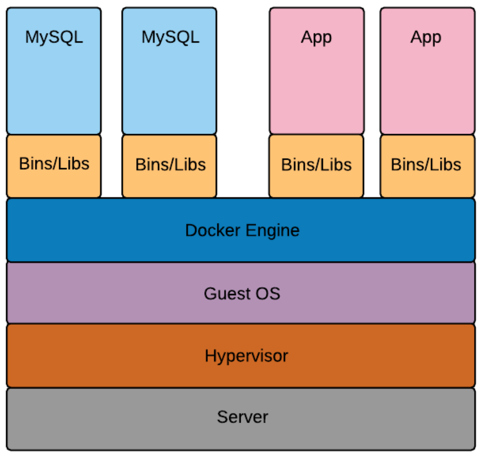
This is where Docker comes in. Docker allows software systems to be packaged and maintained in *images*. Images are templates that describe the software in the package, and, if needed, the software infrastructure (for example, libraries, configuration files, etc.) needed for the software to run.



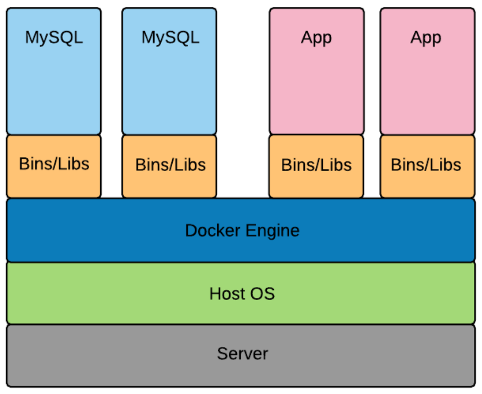
Traditional Systems, aka "Bare Metal"



Virtual Machines



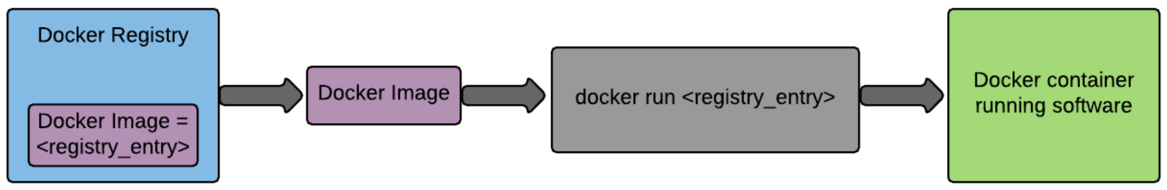
Docker Containers on a Virtual Machine



Docker Containers

What Is Docker?

Docker is an ecosystem that creates running software systems that operate in isolation from each other on a single operating system. In other words, each software package ends up in its own *container*, where it is provided everything it needs to operate, but it cannot interfere with any other software running on the machine. Docker divides up the process of making this happen into images that are stored in *registries*. When Docker wants to run the software system, it takes the description from the image to create a Docker container.


Docker Registry
Docker Image = <registry_entry>
Docker Image
docker run <registry_entry>
Docker container running software

What Is a Docker Image?

A Docker image is a template that describes everything necessary to run a software system, for example, an Apache web server on an Ubuntu Linux distribution. In this example, Docker is instructed to start with an Ubuntu operating system, then install Apache. This communication occurs via a *Dockerfile*. A Dockerfile is a text file containing a series of commands that instruct Docker how to build a software system. The following shows a partial example of a Dockerfile:

```
# Start with Ubuntu image. Install it if necessary

FROM ubuntu

#Who is responsible for this Dockerfile

MAINTAINER Maintenance Guy <main_guy@email.com>


#Run the normal commands to install apache under Ubuntu
RUN apt-get update && apt-get -y install apache2 && apt-get clean

#Set the appropriate environment variables

ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2

# Set the directories

RUN /bin/ln -sf ../sites-available/default-ssl \ /etc/apache2/sites-enabled/001-
default-ssl
RUN /bin/ln -sf ../mods-available/ssl.conf /etc/apache2/mods-enabled/
RUN /bin/ln -sf ../mods-available/ssl.load /etc/apache2/mods-enabled/

#Open the ports Apache uses to receive traffic

EXPOSE 80
EXPOSE 443

CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

This image creates an Apache server running on the Ubuntu Linux distribution on any platform we can run the Docker software (see the Docker installation page for a list of the currently supported platforms).

What Is a Docker Registry?

Docker images are stored in registries. There are two different kinds of Docker registries: public and private. Public registries, such as Docker hub, hold images that can build hundreds of software systems including:

- Apache or Nginx web servers
- MySQL, Cassandra, or MongoDB database servers
- Asterisk or Elasticsearch software packages

Explore the public Docker images on the hub via the search command on any page at https://hub.docker.com.

Private registries hold Docker images that companies want to have complete control over. Companies and other organizations can host their own registries. In addition, Docker can host private registries for organizations that need only a few images stored, or don't want to maintain the registry themselves. (Note: this latter arrangement is very similar to how Github.com, the software control site, manages its hub.)

The Dockerfile shown in the previous section creates an image that causes an Apache server to run on Ubuntu, and it starts by getting an Ubuntu image. If that image is not on the local machine, then the Docker software asks https://hub.docker.com (or a specified registry) to provide the Ubuntu image. Thus, another way to think of a registry is a place where images wait to be invoked.

The newly created image (the end result of the Dockerfile in the previous section) might be stored in a repository on https://hub.docker.com as well. If the repository is called "fake-name-company", then the image might be stored as fake-name-company/apache.

What Is a Docker Container?

So once a software system described by a Docker image has been created, how is that software run? First, install Docker on every machine on which the software will be run. Docker can be installed directly onto most Linux distributions, and onto Windows or Mac OSX systems. The systems Docker supports directly are listed at https://docs.docker.com/installation/.

Next, tell the Docker software to read the image, then to construct a container that runs the software as described. A container is a section of a machine's operating system that is configured to be the run-time environment for a software system. There can be many containers running at once, each sharing the resources of the operating system. Because the containers are isolated from each other, there is no problem with, for example, different versions of the same software using conflicting libraries.

In the case of the Apache system created in the previous sections, it can now run on any computer that has Docker loaded by issuing this command:

```
docker run fake-name-company/apache
```

Now Apache is running in a container.

How Should Docker Be Used?

As shown in the examples, Docker is driven by text commands, either issued on a command line or collected in a text file. This means Docker can integrate with any system that can emit text commands. With the popularity of DevOps increasing, more tools are available to integrate with Docker.

Docker in the Development Phase

Docker makes it easy to give developers private environments in which they develop their software. Docker also makes it easy to automate (and thus come closer to ensuring) that developers thoroughly test their code. For example, Git, the distributed version control system, has the concept of hooks which are scripts that are invoked when designated actions occur, such as committing code to a repository. When a developer checks or commits new code, Git can invoke Docker to start up a new container with which to test that code.

Docker in the Quality Assurance Phase

Docker can be used to create immutable servers, which are software systems that are dedicated to one purpose and are never changed once they are released from development. Updating or changing anything about an immutable server means replacing the server completely.

Testing immutable servers is much less complicated with Docker because there is nothing for the QA department to tweak on its own. The instructions for setting up and running the new software is embedded in the system; just run the `docker` command and test. If the software breaks, send it back to the development team, who can quickly recreate the bug.

Beyond just hunting bugs, it is easy to set up a container that runs a previous version of software, and one that runs the most current version, and to test them on the same machine. Turnaround time for activities like debugging, testing, and A/B testing is minimized.

Docker in the Production Phase

The advantage of immutable servers is that there is nothing to configure, change, and possibly break, because every component is hard-coded into the system. The disadvantage of immutable servers is that if these systems have to be deployed on individual boxes, or even on VMs, the cost and overhead of updates and changes can become onerous.

Using Docker means that an immutable server is just another process running on a box. There is no cost in taking down a container and substituting another, and the only time costs increase is when another physical box, or VM, needs to be added to provide more space for Docker containers. In production, it becomes easy to do things like blue-green deployment, where the existing production network (by default, this is blue), is slowly replaced by new software or a new network (green). As the transition occurs, the production administrators can monitor performance, faults, and hopefully decrease customer complaints. If any indicator signals trouble with the new software, blue floods back into the green areas. Docker allows this to happen quickly because the time required to read a Docker image and start a Docker container is at most seconds, versus the minutes and even tens of minutes required to start a new VM or physical machine.

The Future of Docker

Docker is a powerful engine that can create both small systems that support individual development environments, and complex systems run by large corporations.

Docker and Microservices

Docker is an essential part of the trend toward microservices, where monolithic software systems are being broken up into smaller cooperating services. An example of evolving a system into microservices might be a software system where the login action is broken out from the main body of the program. Because it can be called by many programs, it makes sense to have the login be provided as a separate service that can be tested and certified as secure.

Companies are examining their software architectures to see what other parts can be broken out as services. Independent services can then scale on their own, be updated independently from other services, and allow for frequent software release cycles. See http://microservices.io/ for more information on microservices.

Docker Orchestration

As Docker is called on to work with larger systems, and particularly in the cloud, the complexity of coordinating where containers are running and why increases exponentially. In response, many companies have developed Docker "orchestration" systems to manage this complexity. Because this is a new field with new systems, the rate of change among the offerings is at least monthly, but often even weekly and daily. The following are some of the current Docker orchestration systems:

Kubernetes by Google

Kubernetes manages a cluster of nodes that run containerized applications. This means that Kubernetes can take multiple computers and treat their combined capacity as one large computer on which Docker containers can run. Kubernetes was developed by Google and is used by Google to manage its own infrastructure. Currently, Kubernetes is supported on Google Compute Engine, Rackspace, Microsoft Azure, and vSphere environments.

Elastic Container Service

Like Kubernetes, Elastic Container Service (ECS) manages a cluster of compute instances as a single resource to locate Docker containers. ECS is a free service integrated with the Amazon Web Services (AWS) API infrastructure.

Rancher (http://rancher.com/)

Rancher creates and manages a purpose-built environment for running Docker. Rancher takes computing resources and loads its own software system (RancherOS) to run Docker. Rancher creates a networked cluster environment that provides load balancing, service discovery, and cross-host networking.

Docker

Docker itself provides the following orchestration tools:

- **Machine.** Docker Machine manages the configuration of Docker software on individual computers, or groups of computers (called *swarms* in Docker).
- **Swarm.** Docker Swarm pulls individual computer resources into a single integrated pool of resources that can run Docker containers.
- **Compose.** Docker Compose allows users to take small multiple Docker containers that would support individual services, and deploy those services in a microservice architecture with a single file.

Conclusion

Using Docker allows organizations to create easily deployable software systems that can run on individual or clustered computer systems, on a wide variety of platforms. Docker is platform-agnostic; it can run on bare metal, VMs, or purpose-built systems in data centers, or private, hybrid, or public clouds. Organizations are leveraging Docker to become more agile, responsive, and leaner as they compete in an ever more challenging software environment.

Learn More

Learn more about how you can improve productivity, enhance efficiency, and sharpen your competitive edge through training.

AWS Training

Red Hat® Training

VMware Training

Visit www.globalknowledge.com or call 1-800-COURSES (1-800-268-7737) to speak with a Global Knowledge training advisor.

About the Author

Jon M. Gallagher is a Global Knowledge instructor who has decades of experience creating and running large-scale software systems for the web and for enterprises. He has been using Amazon Web Services since its introduction in 2006.