

Docker: Lightweight Linux Containers for Consistent Development and Deployment

Dirk Merkel

Abstract

Take on “dependency hell” with Docker containers, the lightweight and nimble cousin of VMs. Learn how Docker makes applications portable and isolated by packaging them in containers based on LXC technology.

Imagine being able to package an application along with all of its dependencies easily and then run it smoothly in disparate development, test and production environments. That is the goal of the open-source Docker project. Although it is still not officially production-ready, the latest release (0.7.x at the time of this writing) brought Docker another step closer to realizing this ambitious goal.

Docker tries to solve the problem of “dependency hell”. Modern applications often are assembled from existing components and rely on other services and applications. For example, your Python application might use PostgreSQL as a data store, Redis for caching and Apache as a Web server. Each of these components comes with its own set of dependencies that may conflict with those of other components. By packaging each component and its dependencies, Docker solves the following problems:

- **Conflicting dependencies:** need to run one Web site on PHP 4.3 and another on PHP 5.5? No problem if you run each version of PHP in a separate Docker container.
- **Missing dependencies:** installing applications in a new environment is a snap with Docker, because all dependencies are packaged along with the application in a container.
- **Platform differences:** moving from one distro to another is no longer a problem. If both systems run Docker, the same container will execute without issues.

Docker: a Little Background

Docker started life as an open-source project at dotCloud, a cloud-centric platform-as-a-service company, in early 2013. Initially, Docker was a natural extension of the technology the company had developed to run its cloud business on thousands of servers. It is written in Go, a statically typed programming language developed by Google with syntax loosely based on C. Fast-forward six to nine months, and the company has hired a new CEO, joined the Linux Foundation, changed its name to Docker Inc., and announced that it is shifting its focus to the development of Docker and the Docker ecosystem. As further indication of Docker's popularity, at the time of this writing, it has been starred on GitHub 8,985 times and has been forked 1,304 times. Figure 1 illustrates Docker's rising popularity in Google searches. I predict that the shape of the past 12 months will be dwarfed by the next 12 months as Docker Inc. delivers the first version blessed for production deployments of containers and the community at large becomes aware of Docker's usefulness.

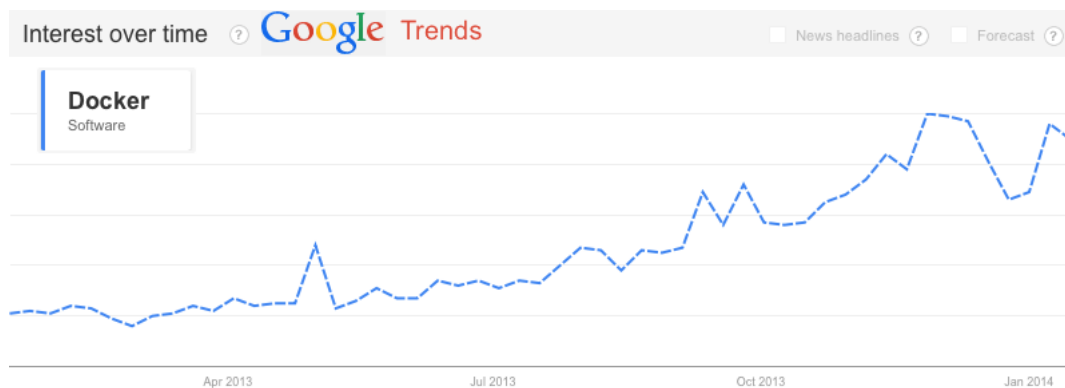


Figure 1. Google Trends Graph for “Docker Software” for Past 12 Months

Under the Hood

Docker harnesses some powerful kernel-level technology and puts it at our fingertips. The concept of a container in virtualization has been around for several years, but by providing a simple tool set and a unified API for managing some kernel-level technologies, such as LXCs (Linux Containers), cgroups and a copy-on-write filesystem, Docker has created a tool that is greater than the sum of its parts. The result is a potential game-changer for DevOps, system administrators and developers.

Docker provides tools to make creating and working with containers as easy as possible. Containers sandbox processes from each other. For now, you can think of a container as a lightweight equivalent of a virtual machine.

Linux Containers and LXC, a user-space control package for Linux Containers, constitute the core of Docker. LXC uses kernel-level namespaces to isolate the container from the host. The user namespace separates the container's and the host's user database, thus ensuring that the container's root user does not have root privileges on the host. The process namespace is responsible for displaying and managing only processes running in the container, not the host. And, the network namespace provides the container with its own network device and virtual IP address.

Another component of Docker provided by LXC are Control Groups (cgroups). While namespaces are responsible for isolation between host and container, control groups implement resource accounting and limiting. While allowing Docker to limit the resources being consumed by a container, such as memory, disk space and I/O, cgroups also output lots of metrics about these resources. These metrics allow Docker to monitor the resource consumption of the various processes within the containers and make sure that each gets only its fair share of the available resources.

In addition to the above components, Docker has been using AuFS (Advanced Multi-Layered Unification Filesystem) as a filesystem for containers. AuFS is a layered filesystem that can transparently overlay one or more existing filesystems. When a process needs to modify a file, AuFS creates a copy of that file. AuFS is capable of merging multiple layers into a single representation of a filesystem. This process is called copy-on-write.

The really cool thing is that AuFS allows Docker to use certain images as the basis for containers. For example, you might have a CentOS Linux image that can be used as the basis for many different containers. Thanks to AuFS, **only one copy of the CentOS image is required**, which results in savings of storage and memory, as well as faster deployments of containers.

An added benefit of using AuFS is Docker's ability to **version container images**. Each new version is simply a **diff of changes from the previous version**, effectively keeping image files to a minimum. But, it also means that you always have a complete audit trail of what has changed from one version of a container to another.

Traditionally, Docker has depended on AuFS to provide a copy-on-write storage mechanism. However, the recent addition of a storage driver API is likely to lessen that dependence. Initially, there are three storage drivers available: AuFS, VFS and Device-Mapper, which is the result of a collaboration with Red Hat.

As of version 0.7, Docker works with all Linux distributions. However, it does not work with most non-Linux operating systems, such as Windows and OS X. The recommended way of using Docker on those OSes is to **provision a virtual machine on VirtualBox using Vagrant**.

Containers vs. Other Types of Virtualization

So what exactly is a container and how is it different from hypervisor-based virtualization? To put it simply, **containers virtualize at the operating system level**, whereas **hypervisor-based solutions virtualize at the hardware level**. While the effect is similar, the differences are important and significant, which is why I'll spend a little time exploring the differences and the resulting differences and trade-offs.

Virtualization:

Both containers and VMs are virtualization tools. On the VM side, **a hypervisor makes siloed slices of hardware available**. There are generally two types of hypervisors: "Type 1" runs directly on the bare metal of the hardware, while "Type 2" runs as an additional layer of software within a guest OS. While the open-source Xen and VMware's ESX are examples of Type 1 hypervisors, examples of Type 2 include Oracle's open-source VirtualBox and VMware Server. Although Type 1 is a better candidate for comparison to Docker containers, I don't make a distinction between the two types for the rest of this article.

Containers, in contrast, **make available protected portions of the operating system**—they effectively **virtualize the operating system**. Two containers running on the same operating system **don't know that they are sharing resources** because each has its own **abstracted networking layer, processes and so on**.

Operating Systems and Resources:

Since hypervisor-based virtualization **provides access to hardware only**, you still need to **install an operating system**. As a result, there are multiple full-fledged operating systems running, one in each VM, which quickly gobbles up resources on the server, such as RAM, CPU and bandwidth.

Containers piggyback on an already running operating system as their host environment. They merely execute in spaces that are isolated from each other and from certain parts of the host OS. This has two significant benefits. First, **resource utilization is much more efficient**. If a container is not executing anything, it is not using up resources, and containers can call upon their host OS to satisfy some or all of their dependencies. Second, containers are **cheap** and therefore **fast to create and destroy**. There is no **need to boot and shut down a whole OS**. Instead, a container merely has to terminate the processes running in its isolated space. Consequently, starting and stopping a container is more akin to starting and quitting an application, and is just as fast.

Both types of virtualization and containers are illustrated in Figure 2.

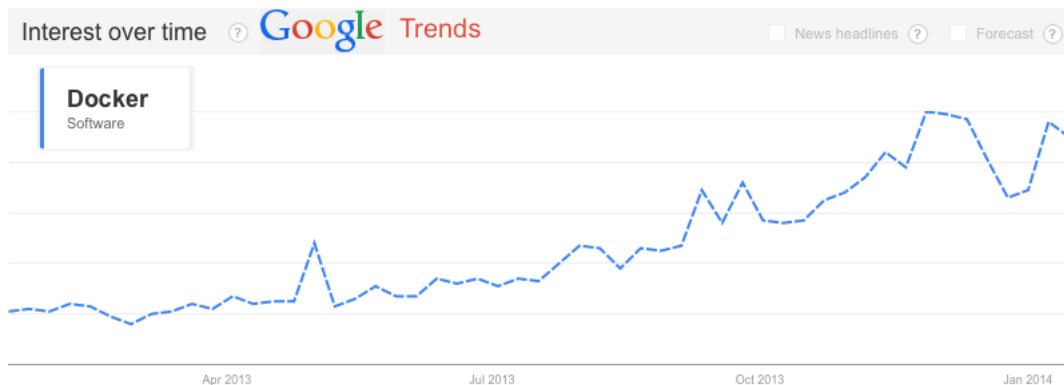


Figure 2. VMs vs. Containers

Isolation for Performance and Security:

Processes executing in a Docker container are isolated from processes running on the host OS or in other Docker containers. Nevertheless, **all processes are executing in the same kernel**. Docker leverages LXC to provide separate namespaces for containers, a technology that has been present in Linux kernels for 5+ years and is considered fairly mature. It also uses Control Groups, which **have been in the Linux kernel even longer**, to implement **resource auditing and limiting**.

The Docker daemon itself also poses a potential attack vector because it currently runs with root privileges. Improvements to both LXC and Docker should allow containers to run without root privileges and to execute the Docker daemon under a different system user.

Although the type of isolation provided is overall quite strong, **it is arguably not as strong as what can be enforced by virtual machines at the hypervisor level**. If the kernel goes down, so do all the containers. The other area where VMs have the advantage is their maturity and widespread adoption in production environments. VMs have been hardened and proven themselves in many different high-availability environments. In comparison, Docker and its supporting technologies have not seen nearly as much action. Docker in particular is undergoing massive changes every day, and we all know that change is the enemy of security.

Docker and VMs—Frenemies:

Now that I've spent all this time comparing Docker and VMs, it's time to acknowledge that these two technologies can actually complement each other. Docker runs just fine on already-virtualized environments. You obviously don't want to incur the cost of encapsulating each application or component in a separate VM, but given a Linux VM, you can easily deploy Docker containers on it. That is why it should not come as a surprise that the officially supported way of using Docker on non-Linux systems, such as OS X and Windows, is to install a Precise64 base Ubuntu virtual machine with the help of Vagrant. Simple detailed instructions are provided on the <http://www.docker.io> site.

The bottom line is that virtualization and containers exhibit some similarities. Initially, it helps to think of containers as very lightweight virtualization. However, as you spend more time with containers, you come to understand the subtle but important differences. Docker does a nice job of harnessing the benefits of containerization for a focused purpose, namely the lightweight packaging and deployment of applications.

Docker Repositories

One of Docker's killer features is the ability to find, download and start container images that were created by other developers quickly. The place where images are stored is called a registry, and Docker Inc. offers a public registry also called the Central Index. You can think of the registry along with the Docker client as the equivalent of Node's NPM, Perl's CPAN or Ruby's RubyGems.

In addition to various base images, which you can use to create your own Docker containers, the public Docker Registry features images of ready-to-run software, including databases, content management systems, development environments, Web servers and so on. While the Docker command-line client searches the public Registry by default, it is also possible to maintain private registries. This is a great option for distributing images with proprietary code or components internally to your company. Pushing images to the registry is just as easy as downloading. It requires you to create an account, but that is free as well. Lastly, Docker Inc.'s registry has a Web-based interface for searching for, reading about, commenting on and recommending (aka "starring") images. It is ridiculously easy to use, and I encourage you to click the link in the Resources section of this article and start exploring.

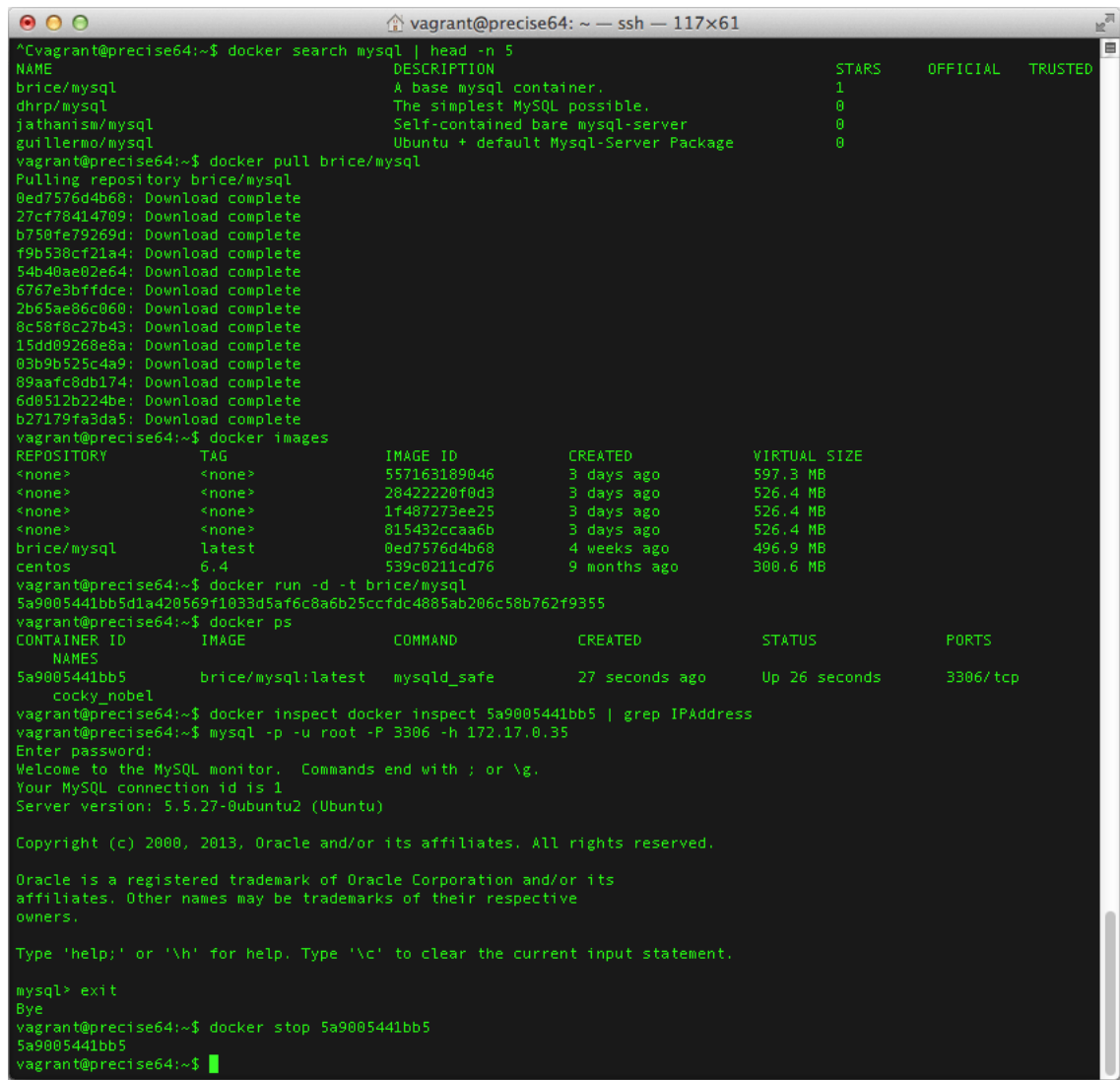
Hands-On with Docker

Docker consists of a single binary that can be run in one of three different ways. First, it can run as a daemon to manage the containers. The daemon exposes a REST-based API that can be accessed locally or remotely. A growing number of client libraries are available to interact with the daemon's API, including Ruby, Python, JavaScript (Angular and Node), Erlang, Go and PHP.

The client libraries are great for accessing the daemon programmatically, but the more common use case is to issue instructions from the command line, which is the second way the Docker binary can be used, namely as a command-line client to the REST-based daemon.

Third, the Docker binary functions as a client to remote repositories of images. Tagged images that make up the filesystem for a container are called repositories. Users can pull images provided by others and share their own images by pushing them to the registry. Registries are used to collect, list and organize repositories.

Let's see all three ways of running the docker executable in action. In this example, you'll search the Docker repository for a MySQL image. Once you find an image you like, you'll download it, and tell the Docker daemon to run the command (MySQL). You'll do all of this from the command line.



```
vagrant@precise64: ~ -- ssh -- 117x61
Cvagrant@precise64:~$ docker search mysql | head -n 5
NAME                DESCRIPTION                STARS   OFFICIAL   TRUSTED
brice/mysql          A base mysql container.    1
dhrp/mysql           The simplest MySQL possible. 0
jathanism/mysql      Self-contained bare mysql-server 0
guillermo/mysql      Ubuntu + default Mysql-Server Package 0
vagrant@precise64:~$ docker pull brice/mysql
Pulling repository brice/mysql
0ed7576d4b68: Download complete
27cf78414709: Download complete
b750fe79269d: Download complete
f9b538cf21a4: Download complete
54b40ae02e64: Download complete
6767e3bffdce: Download complete
2b65ae86c060: Download complete
8c58f8c27b43: Download complete
15dd09268e8a: Download complete
03b9b525c4a9: Download complete
09aafc8db174: Download complete
6d0512b224be: Download complete
b27179fa3da5: Download complete
vagrant@precise64:~$ docker images
REPOSITORY          TAG                IMAGE ID            CREATED             VIRTUAL SIZE
<none>              <none>            557163189046       3 days ago         597.3 MB
<none>              <none>            20422220f0d3       3 days ago         526.4 MB
<none>              <none>            1f487273ee25       3 days ago         526.4 MB
<none>              <none>            815432ccaa6b       3 days ago         526.4 MB
brice/mysql          latest             0ed7576d4b68       4 weeks ago        496.9 MB
centos               6.4               539c0211cd76       9 months ago       300.6 MB
vagrant@precise64:~$ docker run -d -t brice/mysql
5a9005441bb5d1a420569f1033d5af6c8a6b25ccfdc4085ab206c58b762f9355
vagrant@precise64:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
5a9005441bb5        brice/mysql:latest  mysqld_safe         27 seconds ago     Up 26 seconds      3306/tcp
cocky_nobel
vagrant@precise64:~$ docker inspect docker inspect 5a9005441bb5 | grep IPAddress
vagrant@precise64:~$ mysql -p -u root -P 3306 -h 172.17.0.35
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.5.27-0ubuntu2 (Ubuntu)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> exit
Bye
vagrant@precise64:~$ docker stop 5a9005441bb5
5a9005441bb5
vagrant@precise64:~$
```

Figure 3. Pulling a Docker Image and Launching a Container

Start by issuing the `docker search mysql` command, which then displays a list of images in the public Docker registry that match the keyword “mysql”. For no particular reason other than I know it works, let's download the “brice/mysql” image, which you do with the `docker pull brice/mysql` command. You can see that Docker downloaded not only the specified image, but also the images it was built on. With the `docker images` command, you list the images currently available locally, which includes the “brice/mysql” image. Launching the container with the `-d` option to detach from the currently running container, you now have MySQL running in a container. You can verify that with the `docker ps` command, which lists containers, rather than images. In the output, you also see the port on which MySQL is listening, which is the default of 3306.

But, how do you connect to MySQL, knowing that it is running inside a container? Remember that Docker containers get their own network interface. You need to find the IP address and port at which the mysql server process is listening. The `docker inspect <imageId>` provides a lot of info, but since all you need is the IP address, you can just grep for that when inspecting the container by providing its hash `docker inspect 5a9005441bb5 | grep IPAddress`. Now you can connect with the standard MySQL CLI client by specifying the host and port options. When you're done with the MySQL server, you can shut it down with `docker stop 5a9005441bb5`.

It took seven commands to find, download and launch a Docker container to get a MySQL server running and shut it down after you're done. In the process, you didn't have to worry about conflicts with installed software, perhaps a different version of MySQL, or dependencies. You used seven different Docker commands: search, pull, images, run, ps, inspect and stop, but the Docker client actually offers 33 different commands. You can see the full list by running `docker help` from the command line or by consulting the on-line manual.

Before exercising Docker in the above example, I mentioned that the client communicates with the daemon and the Docker Registry via REST-based Web services. That implies that you can use a local Docker client to interact with a remote daemon, effectively administering your containers on a remote machine. The APIs for the Docker daemon, Registry and Index are nicely documented, illustrated with examples and available on the Docker site (see Resources).

Docker Workflow

There are various ways in which Docker can be integrated into the development and deployment process. Let's take a look at a sample workflow illustrated in Figure 4. A developer in our hypothetical company might be running Ubuntu with Docker installed. He might push/pull Docker images to/from the public registry to use as the base for installing his own code and the company's proprietary software and produce images that he pushes to the company's private registry.

The company's QA environment in this example is running CentOS and Docker. It pulls images from the public and private registries and starts various containers whenever the environment is updated.

Finally, the company hosts its production environment in the cloud, namely on Amazon Web Services, for scalability and elasticity. Amazon Linux is also running Docker, which is managing various containers.

Note that all three environments are running different versions of Linux, all of which are compatible with Docker. Moreover, the environments are running various combinations of containers. However, since each container compartmentalizes its own dependencies, there are no conflicts, and all the containers happily coexist.

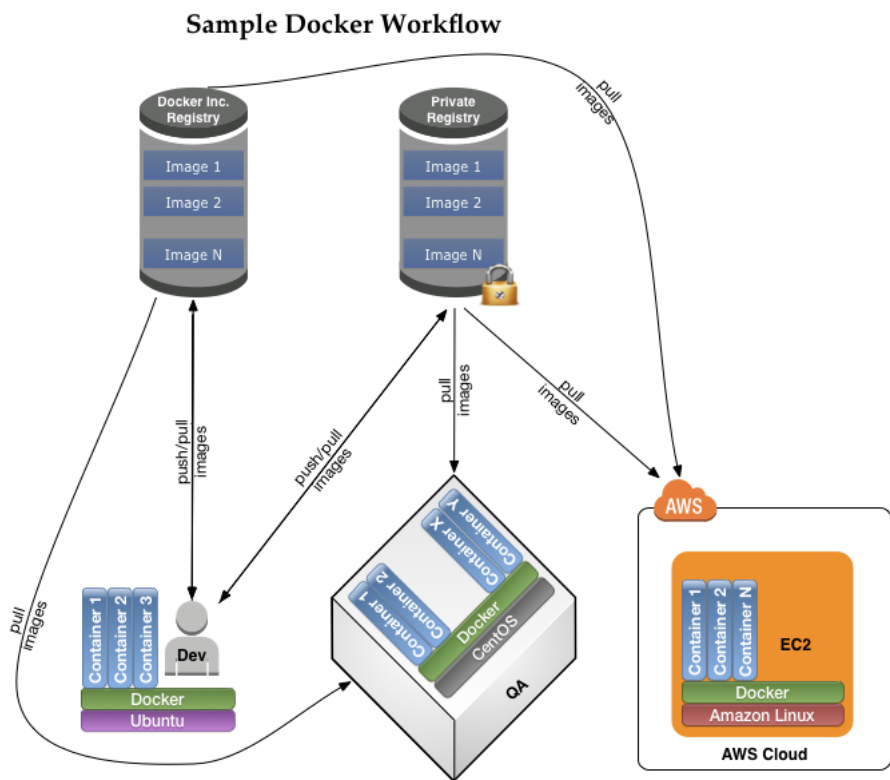


Figure 4. Sample Software Development Workflow Using Docker

It is crucial to understand that Docker promotes an application-centric container model. That is to say, containers should run individual applications or services, rather than a whole slew of them. Remember that containers are fast and resource-cheap to create and run. Following the single-responsibility principle and running one main process per container results in loose coupling of the components of your system. With that in mind, let's create your own image from which to launch a container.

Creating a New Docker Image

In the previous example, you interacted with Docker from the command line. However, when creating images, it is far more common to create a “Dockerfile” to automate the build process. Dockerfiles are simple text files that describe the build process. You can put a Dockerfile under version control and have a perfectly repeatable way of

creating an image.

For the next example, please refer to the “PHP Box” Dockerfile (Listing 1). *Garrick, shrink listing 1.*

Listing 1. PHP Box1160011.qrk

```
# PHP Box # # VERSION 1.0 # use centos base image FROM centos:6.4 # specify the maintainer MAINTAINER Dirk Merkel, dmerkel@vivantech.com # u
```

Let's take a closer look at what's going on in this Dockerfile. The syntax of a Dockerfile is a command keyword followed by that command's argument(s). By convention, command keywords are capitalized. Comments start with a pound character.

The FROM keyword indicates which image to use as a base. This must be the first instruction in the file. In this case, you will build on top of the latest CentOS base image. The MAINTAINER instruction obviously lists the person who maintains the Dockerfile. The RUN instruction executes a command and commits the resulting image, thus creating a new layer. The RUN commands in the Dockerfile fetch configuration files for additional repositories and then use Yum to install curl, git, wget, unzip, httpd, php-mysql and yum-utils. I could have combined the yum install commands into a single RUN instruction to avoid successive commits.

The EXPOSE instruction then exposes port 80, which is the port on which Apache will be listening when you start the container.

Finally, the CMD instruction will provide the default command to run when the container is being launched. Associating a single process with the launch of the container allows you to treat a container as a command.

Typing `docker build -t php_box .` on the command line will now tell Docker to start the build process using the Dockerfile in the current working directory. The resulting image will be tagged “php_box”, which will make it easier to refer to and identify the image later.

The build process downloads the base image and then installs Apache httpd along with all dependencies. Upon completion, it returns a hash identifying the newly created image. Similar to the MySQL container you launched earlier, you can run the Apache and PHP image using the “php_box” tag with the following command line: `docker run -d -t php_box`.

Let's finish with a quick example that illustrates how easy it is to layer on top of an existing image to create a new one:

```
# MyApp # # VERSION 1.0 # use php_box base image FROM php_box # specify the maintainer MAINTAINER Dirk Merkel, dmerkel@vivantech.com # put m
```

This second Dockerfile is shorter than the first and really contains only two interesting instructions. First, you specify the “php_box” image as a starting point using the FROM instruction. Second, you copy a local directory to the image with the ADD instruction. In this case, it is a PHP project that is being copied to Apache's DOCUMENT_ROOT folder on the images. The result is that the site will be served by default when you launch the image.

Conclusion

Docker's prospect of lightweight packaging and deploying of applications and dependencies is an exciting one, and it is quickly being adopted by the Linux community and is making its way into production environments. For example, Red Hat announced in December support for Docker in the upcoming Red Hat Enterprise Linux 7. However, Docker is still a young project and is growing at breakneck speed. It is going to be exciting to watch as the project approaches its 1.0 release, which is supposed to be the first version officially sanctioned for production environments. Docker relies on established technologies, some of which have been around for more than a decade, but that doesn't make it any less revolutionary. Hopefully this article provided you with enough information and inspiration to download Docker and experiment with it yourself.

Docker Update11600s1.qrk

As this article was being published, the Docker team announced the release of version 0.8. This latest deliverable adds support for Mac OS X consisting of two components. While the client runs natively on OS X, the Docker daemon runs inside a lightweight VirtualBox-based VM that is easily managed with boot2docker, the included command-line client. This approach is necessary because the underlying technologies, such as LXC and name spaces, simply are not supported by OS X. I think we can expect a similar solution for other platforms, including Windows.

Version 0.8 also introduces several new builder features and experimental support for BTRFS (B-Tree File System). BTRFS is another copy-on-write filesystem, and the BTRFS storage driver is positioned as an alternative to the AuFS driver.

Most notably, Docker 0.8 brings with it many bug fixes and performance enhancements. This overall commitment to quality signals an effort by the Docker team to produce a version 1.0 that is ready to be used in production environments. With the team committing to a monthly release cycle, we can look forward to the 1.0 release in the April to May timeframe.

Resources11600s2.qrk

Main Docker Site: <https://www.docker.io>

Docker Registry: <https://index.docker.io>

Docker Registry API: http://docs.docker.io/en/latest/api/registry_api

Docker Index API: http://docs.docker.io/en/latest/api/index_api

Docker Remote API: http://docs.docker.io/en/latest/api/docker_remote_api

11600s3.qrk

Send comments or feedback via <http://www.linuxjournal.com/contact> or to <ljeditor@linuxjournal.com>.