文献翻译

|  |  |  |
| --- | --- | --- |
| 题 目 | 基于Nodejs的社团 | |
|  | 集成工具式管理系统 | |
| 学生姓名 | 张超 | |
| 专业班级 | 电子信息科学与技术14-01 | |
| 学 号 | 54140720149 | |
| 院 （系） | 计算机与通信工程学院 | |
| 指导教师（职称） | | 陈冬冬（讲师） |
| 完成时间 | 2017年4月18日 | |
|  |  | |

Docker: 软件即服务，一个操作系统级别的虚拟机框架

John Fink

(麦克马斯特大学数字奖学金馆员)

摘 要：Docker 是本地用于64位 Linux 的相对较新的虚拟化方法。与更传统的虚拟化技术相比，Docker 在系统资源方面更轻，提供类似 git 的提交和标签系统，并且可以从笔记本电脑扩展到云。

关键词：虚拟化；自建；定制；Docker

1. 简述

Docker 是本地用于64位 Linux 的相对较新的虚拟化方法。与更传统的虚拟化技术相比，Docker 在系统资源方面更轻，提供类似 git 的提交和标签系统，并且可以从笔记本电脑扩展到云。

1. 介绍

如果你过去几年从事过软件库工作，你可能会记得你的服务器机房的模样。运行 Novell 网络操作系统的机箱连接到庞大的多盘 CDROM 阵列。冰箱大小的太阳盒子。数字的 AlphaServers 运行你的软件库目录。为了运行大部分软件库的重要业务，你需要认真的部署，结果，机房经常混杂着大量的书架，电线和空调设备。随着运行在微型计算机上的 Linux 的出现，这些房间变得稍微小一些，也许稍微不那么复杂，但直到 21 世纪初，服务器机房才真正发生了巨变， 快速采用易于实现的虚拟化，它允许在一台机器上运行多个分立的操作系统。

尽管现代意义上的虚拟化实际上早在 20 世纪 60 年代就发生在利用 360 台机器和 IBM 系统的 VM 上，而且 28 6和 386 芯片都包含了一些虚拟化技术，但直到 2001 年 VMWare 推出其基于 Linux 在 x86 虚拟化产品的成功，虚拟化才真正受到人们关注。由于像 KVM 和 Xen 这样的技术在本地部署和诸如亚马逊网络服务和 OpenStack 框架这样的声称云计算的产品也日渐崛起，混乱的白色 PC 机箱还有利用 CDROM 塔式服务器机房或多或少已经灭绝。今天，你已经可以使用最少的物理机创建多个虚拟机。

virsh # list

Id Name State

----------------------------------

1 trotsky running

2 eris running

3 funhouse running

4 gorgar running

5 shoah running

6 balder2 running

7 asgard running

在作者工作场所的单个服务器上列出虚拟机。

1. 库上的虚拟化

现代虚拟化对软件库的需求因机构不同而有很大差异。一个中小型的IT商城不会做大量的软件开发工作 - 正如许多库中传统的那样 - 可能只需要少量的虚拟机。例如，它可能需要一个已经在网络上有的库，一个用于库管理系统，另一个用于控制用于客户的工作站的镜像。尽可能地，这些将成为那些老式白色机盒的替代品。某些情况下，像给定的库管理系统一样，会受到约定或许可的约束，以尽可能少地偏离传统安装。对于其他人来说，定制安装到虚拟化框架的努力可能不值得付出;在性能或便利方面的收益可能很小。

有开发软件人员的图书馆可能有更复杂的要求。现代的开发环境是不断迭代的。编写代码。编写测试。代码中断。写更多的代码。写更多的测试。打破更多的事情。想象一下，每次你想从头开始测试你的代码时，手动设置一台新机器，甚至是一台虚拟机器。任何使这个过程更高效的事情都可以让开发人员花更多时间完成他们擅长的特定工作。例如，流行的 Virtualbox 软件非常重视面向 GUI，因此已经构建了一个完整的框架，用于创建和销毁 Virtualbox VMs 脚本。

典型的图书馆虚拟化方案主要利用机器级虚拟化。已经提到的系统 - KVM，VMware，Xen，Virtualbox 以及 DOSBox（专门用于运行 DOS 游戏的多平台模拟器）等产品都是机器级仿真器。他们试图尽可能多地模拟软件中的计算环境，直到磁盘驱动器，RAM 分配，图形，硬盘驱动器空间甚至处理器类型。例如，一些机器级虚拟化平台可以在基于 Intel 的平台（如台式PC）上仿真基于 ARM 的系统，如 Raspberry Pi。但是这种类型的虚拟化可能是资源密集型和效率低下的。当在一台计算机上运行多个虚拟机时，您可能会通过分配大型磁盘驱动器或为每个 VM 实例分配多个千兆字节的专用 RAM 来快速运行物理机器的限制。这些问题的解决方法相对比较简单（例如将外部驱动器安装为网络共享），但其他问题则更加困难（如处理有限的RAM）。

进入操作系统级别的虚拟化，该虚拟化尝试在实例之间共享资源，而不是模拟尽可能多的实际机器。典型的操作系统级虚拟化方案将共享 RAM，磁盘空间和内核与客户实例。因此，与同等数量的机器级虚拟化实例相比，任意数量的操作系统级虚拟化实例不太可能耗尽主机系统资源。但是这种灵活性需要付出代价。由于来宾实例必须共享一个内核，因此需要共享一个处理器和操作系统类型，因此无法在 Linux 主机下的 x86 或 Windows 上运行该虚拟 Raspberry Pi。尽管存在这些限制，但由于操作系统级别的虚拟化易于部署和轻量级，因此它正成为开发工作中非常有吸引力的工作流程选项。

1. 使用 Docker 进行虚拟化

Docker 是开源操作系统级虚拟化的开源实现，正在迅速获得开发人员的关注。像大多数优秀的开源项目一样，Docker 结合了许多现有的 Linux 技术和新功能;它使用已经存在的技术，如写入时复制联合文件系统（通常为 AUFS）和 Linux 容器，并将它与许多使其成为以开发人员为中心的功能相结合（因此与传统虚拟机不同试图尽可能多地嘲讽机器的隐喻）：像部署可移植性，版本化，重用和可重复性。

这些功能值得考虑。他们将虚拟机配置的概念从耗时的以系统管理员为中心的模型转变为更注重于面向开发人员的工作流程，具体表现在 Docker 版本系统的 git 类本质与它的差异和标签）。

Docker 的核心是一个虚拟化框架，专注于运行应用程序，而不是模拟硬件，这看起来很容易，但强调了 Docker 等操作系统级虚拟化软件与机器级虚拟化之间的关键区别。机器级别的虚拟机关于忠实的硬件重建 - 直到 RAM 分配，分配多少 CPU，模拟 NIC 等等，而操作系统级别的虚拟化是关于应用程序，而不是机器。大多数情况下，当我们开发，测试和发布软件时，我们关心应用程序，而不是真正的或者虚拟的 - 我们正在开发的特定硬件环境，可能除了模拟历史硬件和其他边缘案例。当我们写邮件列表寻求Drupal的帮助时，我们没有说“我有一台带有4个 Intel Core2 Duo 处理器的 Dell PowerEdge 5100，16GB 的 70ns 内存和两块 Atheros 100GB NIC 卡，我无法将此 Drupal 模块正确地工作。“除非我们有一个相当强的指标，说明我们的问题是硬件绑定的，否则我们专注于软件。碰巧，Docker 也是这样做的。通常没有理由尝试为 Docker 定义应该具有多少内存，硬盘应该多大，或者占用多少 CPU，您通常不会为您在桌面上编写的代码执行此操作。

在实践中，这意味着运行 Docker 的计算机可以运行比运行典型虚拟机的同一台计算机更多的同时运行的虚拟实例。为了说明：在2007年的一台贫血台式机上与机器级虚拟化搏斗时，我无法同时运行两个 Virtualbox 实例。但是，在同一台主机上，我能够运行超过40个 Docker 容器，而不会明显降低主机系统性能。

1. 总结

Docker 经历了一年多的沉重的开发，在那段时间里，它的创建者或多或少地阻止人们试图将Docker 作为生产就绪框架来运行。然而，在 6 月 9 日，Docker 达到了1.0 的状态，因此应该考虑准备好生产实例。重要的是，Docker 1.0 与所有先前版本的 Docker 兼容，所以当 Docker 成为移动目标时开始的项目仍然可以继续使用。

然而，将更深奥的应用程序移植到 Docker 并不是一个简单的过程。 Docker 希望在前台进程中运行，因此有必要将常用程序（如 MySQL 和 Apache ）从通常的后台模式转换为前台模式，并且 Docker 专注于每个容器的一个应用程序（通过 docker-wordpress 和许多其他 Docker 应用程序通过审慎使用 supervisord ）使运行中的产品具有复杂的安装过程，这种过程略低于最佳状态，现在可能更适合于传统的虚拟机或裸机部署。

尽管 Docker 还有很多工作要做，但它正在迅速接近稳定状态，并显示出将其纳入图书馆的虚拟化活动的明确承诺。对于软件开发，当程序员检查他们的代码到 git 中时，Dockerfile 可以包含在源代码中，允许在远程服务器上快速测试代码，或者作为一个演示工具，让其他人快速创建他们自己的应用程序版本，而无需担心特定的建筑指令或依赖管理。 Docker可以用作备份策略 - 作为图书馆网站备份脚本的一部分，Docker 映像可以在服务中断时快速部署。对于常规的系统管理任务，谁不愿意看到一个巨大的安装程序归结为一个单一的命令？

参考文献

1. Docker J. Postmodernism and popular culture: A cultural history[M]. Cambridge University Press, 1994.
2. Docker B B, Hubble T C T. Quantifying root-reinforcement of river bank soils by four Australian tree species[J]. Geomorphology, 2008, 100(3-4): 401-418.
3. Semmens L T, France R B, Docker T W G. Integrated structured analysis and formal specification techniques[J]. The Computer Journal, 1992, 35(6): 600-610.
4. Docker J. The Nervous Nineties: Australian cultural life in the 1890s[M]. Oxford University Press Australia, 1991.
5. Therriault T W, Docker M F, Orlova M I, et al. Molecular resolution of the family Dreissenidae (Mollusca: Bivalvia) with emphasis on Ponto-Caspian species, including first report of Mytilopsis leucophaeata in the Black Sea basin[J]. Molecular phylogenetics and evolution, 2004, 30(3): 479-489.

[Issue 25, 2014-07-21](https://journal.code4lib.org/issues/issues/issue25)

Docker: a Software as a Service, Operating System-Level Virtualization Framework

Docker is a relatively new method of virtualization available natively for 64-bit Linux. Compared to more traditional virtualization techniques, Docker is lighter on system resources, offers a git-like system of commits and tags, and can be scaled from your laptop to the cloud.

By John Fink, Digital Scholarship Librarian, McMaster University

**Introduction**

If you were working in library IT in the last millennium, you’ll likely remember what your server room used to look like. PC towers running Novell Netware attached to huge multi-disc CDROM arrays. Refrigerator-sized Sun boxes. Digital AlphaServers running your library catalogue. To run most of the serious business of libraries, you needed serious equipment, and, as a result, machine rooms were often jumbled messes of shelves, wires and air conditioning units. With the advent of Linux running on microcomputers, these rooms became slightly smaller and maybe slightly less complex, but it wasn’t until the early 2000s that the real sea change came for the server room — the rapid adoption of easily implementable virtualization [[1](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note1)], which allowed running multiple discrete operating systems on a single machine.

Although virtualization in the modern sense actually happened as early as the 1960s with VMs for IBM System/360 machines [[2](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note2)], and both the 286 and 386 chips contained some species of virtualization [[3](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note3)], it wasn’t until 2001 when VMWare introduced its x86 virtualization products that Linux-based virtualization really took off. With technologies like KVM [[4](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note4)] and Xen [[5](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note5)] looming large for local deployment and products like Amazon Web Services [[6](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note6)] and the OpenStack [[7](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note7)] framework claiming the cloud, the jumbled, messy, white-box-PC-and-CDROM-tower-laden server room is more or less extinct; today, you can create multiple virtual machines out of minimal physical machines.

virsh # list

Id Name State

----------------------------------

1 trotsky running

2 eris running

3 funhouse running

4 gorgar running

5 shoah running

6 balder2 running

7 asgard running

A listing of virtual machines on a single server at the author’s workplace.

**Virtualization in libraries**

Modern virtualization needs for libraries vary greatly from institution to institution. A small to midrange IT shop that doesn’t do a whole lot of software development — as what you’d find traditionally in many libraries — might only require a handful of virtual machines. It may need one for the library’s web presence, one for the ILS, and one to control images for patron workstations, for example. As much as possible, these would be drop-in replacements for those old white boxes. Some instances, like for a given ILS, would be bound by convention or licensing to have as little deviation from a traditional install as possible. For others, the effort to customize an install to a virtualization framework might not be worth the payoff; gains in performance or convenience may very well be minimal.

Libraries with staff who develop software likely have more complex requirements. The modern development environment is one of constant iteration. Write code. Write tests. Code breaks. Write more code. Write more tests. Break more things. Imagine manually having to set up a new machine, even a virtual one, each time you wanted to test your code from scratch. Anything that makes this process more efficient lets developers spend more time doing the specific work that they’re good at. The popular Virtualbox [[8](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note8)] software, for instance, is very heavily GUI-oriented [[9](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note9)] and has therefore had an entire framework [[10](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note10)] built around it designed to make the creating and destroying of Virtualbox VMs scriptable.

Typical library virtualization schemes mostly utilize *machine-level virtualization*. The systems already mentioned — KVM, VMware, Xen, Virtualbox, along with products like DOSBox [[11](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note11)], a multiplatform emulator specifically written to run DOS games — are machine-level emulators. They attempt to emulate as much as possible about a computing environment in software, down to disk drives, RAM allocation, graphics, hard drive space, and even processor type. Some machine-level virtualization platforms, for instance, make it possible to emulate an ARM-based system like a Raspberry Pi [[12](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note12)] on an Intel-based platform like a desktop PC. But this type of virtualization can be resource-intensive and inefficient. When running multiple virtual machines on a single computer, you might quickly run into the limits of the physical machine by allocating large disk drives or carving out multiple Gigabytes of dedicated RAM for each VM instance. Workarounds for some of these problems are relatively simple (like mounting external drives as network shares), but others are more difficult (such as dealing with limited RAM).

Enter operating system-level virtualization [[13](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note13)], which tries to share resources amongst instances rather than emulate as much of an actual machine as possible. Typical operating system-level virtualization schemes will share RAM, disk space, and kernel with guest instances. Consequently, an arbitrary number of operating system-level virtualization instances will be less likely to run out of host system resources than an equivalent number of machine-level virtualization instances. But that flexibility comes at a cost. Because guest instances must share a kernel and therefore both a processor and operating system type [[14](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note14)], you could not run that virtual Raspberry Pi on x86 or Windows under a Linux host. Despite these limitations, operating system level virtualization is emerging as a very attractive workflow option for development work due to its ease of deployment and lightweight nature.

**Virtualization with Docker**

Docker [[15](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note15)], an open-source implementation of operating system-level virtualization, is rapidly gaining mindshare amongst developers. Like most good open source projects, Docker incorporates a lot of existing Linux technologies along with new functionality; it uses already existing technologies like copy-on-write union filesystems (usually AUFS [[16](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note16)]) and Linux Containers [[17](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note17)], and it couples that with a number of features that make it developer-centric (and therefore distinct from traditional virtual machines that attempt to hew as much as possible to the metaphor of *machine*): like deployment portability, versioning, re-use, and repeatability [[18](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note18)].

These features are worth considering. They transform the notion of virtual machine provisioning from a time-consuming, sysadmin-centric model to one focused more on a developer-oriented workflow, [[19](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note19)] as evidenced in particular by the git [[20](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note20)]-like nature of Docker’s versioning system (with its diffs and tags).

At its core, Docker is a virtualization framework focused around running applications and not around emulating hardware, which seems facile at first but underscores the critical difference between operating system-level virtualization software like Docker and machine-level virtualization. Machine-level VMs are about faithful recreation of hardware — right down to RAM allotment, how many CPUs to assign, emulating NICs, and so forth — and operating system level virtualization is about *applications*, not machines [[21](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note21)]. Most of the time when we’re developing, testing, and releasing software we care about applications and not really the specific hardware environment — real or virtual — that we’re developing in, perhaps except with emulating historical hardware [[22](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note22)] and other edge cases. When we write to mailing lists asking for help with Drupal, we don’t say “I have a Dell PowerEdge 5100 with 4 Intel Core2Duo processors and 16GB of 70ns RAM and two Atheros 100GB NIC cards and I can’t get this Drupal module to work correctly.” Unless we have a pretty strong indicator that our problem is hardware bound, we focus on software. As it happens, this is what Docker does also. There is usually no reason to try to define for Docker how much memory it should have, how large its hard drive should be, or how much CPU it should take up; you don’t generally do this for the code you write on your desktop either.

In practice, this means a computer running Docker can run many more simultaneous virtual instances than the same computer running typical virtual machines. To illustrate: while wrestling with machine-level virtualization on an anemic desktop dating from 2007, I had trouble running two Virtualbox instances concurrently. But, on the same host, I was able to run over 40 Docker containers without a noticeable drop in host system performance.

**Running Docker**

What does it look like to run Docker? After the Docker software is installed [[23](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note23)], you’re left with a primary binary (“docker”) with which you can start, stop, import, export, and do other [[24](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note24)] operations.

Here’s an example of a Docker host running a few containers, which we can see by running docker ps on the comand line.

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

1bc191f4cdbb bedework:latest supervisord -n 11 days ago Up 11 days 0.0.0.0:8080->8080/tcp sick\_newton

b58946da298c papyrus-demo:port6000 /bin/bash 13 days ago Up 13 days 3000/tcp, 6000/tcp, 0.0.0.0:6001->6001/tcp drunk\_bell

c90c0a6be88f saucy-csclub:latest /bin/bash 13 days ago Up 13 days 0.0.0.0:9999->9999/tcp angry\_shockley

e5a0f8a71f7e papyrus-demo:in-process /start.sh 2 weeks ago Up 13 days 0.0.0.0:3000->3000/tcp mad\_poincare

f752161937c6 ldap\_update\_pw:latest supervisord -n 5 weeks ago Up 13 days 0.0.0.0:5000->5000/tcp distracted\_nobel

33cf9eb89073 catmandu:in-process /bin/bash 6 weeks ago Up 13 days cranky\_mccarthy

Individual docker instances are split up into *images* and *containers*. Containers run instances of images. You can have several containers that come from the same image or variants of that image. In the above list, we can see that container id b58946da298c and e5a0f8a71f7e are running versions of the image “papyrus-demo,” demonstrating that images can have tags that act similarly to git tags, representing different states of a common image. In the papyrus-demo’s image, there’s a tag “in-process” and a tag “port6000”; an image without a distinct tag is always “latest”.

**A simple example: Article-as-container**

To demonstrate how Docker works, I’ve created a container that converts the Markdown that this article was originally written in into HTML via pandoc [[25](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note25)] and serves it using the Python SimpleHTTPServer module. You can get the Docker container from the Github repository [[26](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note26)] for this article.

It’s a very simple Docker container and is built with two components. The first of which, the Dockerfile, runs when the image is first built and the second, start.sh, runs at each instantiation of image into container. The article text is in c4l-docker-article.md, and it’s this file that gets converted by pandoc and served with SimpleHTTPServer in start.sh.

Let’s break down the components and how to run them.

Dockerfile

We’ll start by looking at the Dockerfile line by line.

FROM ubuntu:latest

This is the image to use as a base. In this case, it’s building from Docker’s stock Ubuntu image.

MAINTAINER John Fink <john.fink@gmail.com>

The maintainer of the created image.

RUN echo "deb http://archive.ubuntu.com/ubuntu precise universe" >> /etc/apt/sources.list

The RUN statement runs a command from within the image being built. In this case, it’s adding the Universe repository to Ubuntu’s source list, so later on we can install pandoc.

RUN DEBIAN\_FRONTEND=noninteractive apt-get update

RUN DEBIAN\_FRONTEND=noninteractive apt-get -y install python git pandoc

Two RUN statements, updating Ubuntu and installing the necessary packages.

ADD ./start.sh /start.sh

ADD inserts files from outside the building image. Here, start.sh is added to the image.

RUN mkdir /article/

ADD ./c4l-docker-article.md /article/c4l-docker-article.md

A RUN and an ADD  
together, making the directory and adding the article Markdown to it.

EXPOSE 8888

EXPOSE tells Docker that containers built from the image will have programs running that need access to port 8888; in this case, for Python’s SimpleHTTPServer.

CMD ["/bin/bash", "/start.sh"]

The CMD statement defines a default program to run inside the container if no specific command is given; here, we want containers to run the start.sh file using /bin/bash.

**start.sh**

The extremely basic start.sh files looks like this:

cd /article/

pandoc c4l-docker-article.md -o index.html

python -m SimpleHTTPServer 8888

All it does is go to the article’s directory, convert the article from Markdown to html, and then spawn Python’s SimpleHTTPServer to serve the article.

**Building the article container**

Running the docker build command builds the container. The command parses each line of the Dockerfile, saving each step as a commit and layering the next commit on top. This makes it easy to roll back if errors are detected. As a performance bonus, subsequent invocations of docker build will use the prior layers as a cache, only building new layers when a line is changed. This usually makes building images from edits very fast.

docker build -t c4l-docker-wordpress git://github.com/jbfink/c4l-docker-article

And after the build, launch the container with:

docker run -Pd c4l-docker-article

Every time a new container is created from the c4l-docker-article image — as we do when we issue the docker run command — the CMD statement from the Dockerfile is executed; in this case, our start.sh file. This file converts the article markdown to HTML and runs SimpleHTTPServer to serve the file. The flag P tells Docker to expose a random port on the host to the container port 8888 so SimpleHTTPServer can be reached from outside the container, and flag d tells Docker to run the container detached in the background. Running docker ps should show a line with the new container and the random port assigned to it; going to the Docker host at that port with any web browser renders the article.

A real-world example: WordPress

In the spring of 2013 I started building a Docker WordPress container with an eye towards using it in-house for development projects. Why WordPress? WordPress is the white lab rat of library software — it’s used everywhere, is well supported, is well understood, is generally easy to take care of, and has a huge host of ancillary software behind it. It’s a good real-world example for testing Docker’s capabilities. Initially I built the container manually — that is, by launching a single Docker container running a bash shell and basically performing the steps in the above Dockerfile by hand (doing the normal apt-gets and vim editing of config files). I put it up on docker index [[27](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note27)] and was contacted by a few folks in email about how I built it. In August of 2013 I started work on docker-wordpress [[28](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note28)], a structured way of building what I had done manually that people could play with and build on. The key problem with setting up WordPress in a normal fashion, freezing it in a Docker image, and then running that elsewhere is that the configuration would remain the same across containers — same MySQL passwords and WordPress salts and keys in PHP. Ideally every time docker-wordpress is run there should be different values for all the fiddly WordPress configuration options, so docker-wordpress contains start.sh [[29](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note29)], which runs a series of commands at first inception to set values for things like salts in wp-config.php.

sed -e "s/database\_name\_here/$WORDPRESS\_DB/

s/username\_here/$WORDPRESS\_DB/

s/password\_here/$WORDPRESS\_PASSWORD/

/'AUTH\_KEY'/s/put your unique phrase here/`pwgen -c -n -1 65`/

/'SECURE\_AUTH\_KEY'/s/put your unique phrase here/`pwgen -c -n -1 65`/

/'LOGGED\_IN\_KEY'/s/put your unique phrase here/`pwgen -c -n -1 65`/

/'NONCE\_KEY'/s/put your unique phrase here/`pwgen -c -n -1 65`/

/'AUTH\_SALT'/s/put your unique phrase here/`pwgen -c -n -1 65`/

/'SECURE\_AUTH\_SALT'/s/put your unique phrase here/`pwgen -c -n -1 65`/

/'LOGGED\_IN\_SALT'/s/put your unique phrase here/`pwgen -c -n -1 65`/

/'NONCE\_SALT'/s/put your unique phrase here/`pwgen -c -n -1 65`/" /var/www/wp-config-sample.php > /var/www/wp-config.php

This ensures that different values for important variables get loaded each time docker-wordpress is first run. Combining this with the output from the initial build [[30](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note30)] from source means a fairly long startup time, but subsequent runs usually take less than 30 seconds, and rebuilds (which can be cached) really only need to happen when major updates to component software happen. Running the docker-wordpress image is a fairly simple affair, and new containers take about a second to spawn, after which they can be accessed via internal IPs or given outward-facing ports on the host machine.

Docker-wordpress has a great advantage in that it’s one image, can be run in one container, and is easy to understand. But it would be a mistake to consider it a good model for a production-type instance. In particular, its slapdash approach to logging (something Logstash [[31](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note31)] could go a long way towards fixing) and inclusion of a local MySQL instance make it a difficult sell for a production environment as it is currently written. Consider running 20 docker-wordpresses, each with its own database; it would make much more sense to have a single MySQL server serving multiple WordPress instances.

**Conclusion**

Docker has been under heavy development for over a year, and during that time its creators have more or less discouraged people from attempting to run Docker as a production-ready framework. However, on June 9th, Docker reached 1.0 [[32](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note32)] status, and as such should be considered ready for production instances. Importantly, Docker 1.0 is compatible with all prior versions of Docker, so projects that started when Docker was a moving target can still be used going forward.

However, porting more esoteric applications to Docker is not yet an easy procedure. Docker wants to run things in foreground processes, making it necessary to convert common programs like MySQL and Apache from their usual background modes to foreground ones, and Docker’s focus on one application per container (achieved in docker-wordpress and many other Docker applications through judicious use of supervisord) makes running products with complicated install procedures somewhat less than optimal and perhaps more suited to traditional VM or bare iron deployments for now.

Despite the work yet to be done on Docker, it is rapidly approaching a stable state [[32](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note32)] and shows definite promise for incorporating into libraries’ virtualization activities. For software development, when programmers check their code into git, a Dockerfile could be included in the source code, allowing for quick testing of code on remote servers or as a demonstration tool to let others quickly bring up their own versions of an application without having to worry about specific building instructions or dependency management. Docker could be used as a backup strategy — as part of a backup script for a library’s web site, a Docker image could be built for a rapid deployment in case of service outage. And for regular systems administration tasks, who wouldn’t be happy to see a huge installation procedure [[33](http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+#note33)] boiled down to a single command?

**Endnotes**

1. Docker J. Postmodernism and popular culture: A cultural history[M]. Cambridge University Press, 1994.
2. Docker B B, Hubble T C T. Quantifying root-reinforcement of river bank soils by four Australian tree species[J]. Geomorphology, 2008, 100(3-4): 401-418.
3. Semmens L T, France R B, Docker T W G. Integrated structured analysis and formal specification techniques[J]. The Computer Journal, 1992, 35(6): 600-610.
4. Docker J. The Nervous Nineties: Australian cultural life in the 1890s[M]. Oxford University Press Australia, 1991.
5. Therriault T W, Docker M F, Orlova M I, et al. Molecular resolution of the family Dreissenidae (Mollusca: Bivalvia) with emphasis on Ponto-Caspian species, including first report of Mytilopsis leucophaeata in the Black Sea basin[J]. Molecular phylogenetics and evolution, 2004, 30(3): 479-489.

**About the author**

John Fink is the Digital Scholarship Librarian in the Sherman Centre for Digital Scholarship at McMaster University in Hamilton, Ontario. He has been working with open source and libraries since 1995.