Write a Memory Game with Bash

What's Next for Python

weBoost 4G-X OTR Review

LINUX JOURNAL

Since 1994: The or[illegible] of the Lin[illegible] community

CONTAINERS

Linux Control Groups and Process Isolation | Working with LXC
Orchestration with Kubernetes | GUI Options for Docker

ISSUE 289 | AUGUST 2018
www.linuxjournal.com

CONTENTS

AUGUST 2018
ISSUE 289

72 DEEP DIVE: Containers

CONTENTS

LINUX JOURNAL (ISSN 1075-3583) is published monthly by Linux Journal, LLC. Subscription-related correspondence may be sent to 9597 Jones Rd, #331, Houston, TX 77065 USA. Subscription rate is $34.50/year. Subscriptions start with the next issue.

ARTICLES

AT YOUR SERVICE

SUBSCRIPTIONS: *Linux Journal* is available as a digital magazine, in PDF, EPUB and MOBI formats. Renewing your subscription, changing your email address for issue delivery, paying your invoice, viewing your account details or other subscription inquiries can be done instantly online: http://www.linuxjournal.com/subs. Email us at subs@linuxjournal.com or reach us via postal mail at *Linux Journal*, 9597 Jones Rd #331, Houston, TX 77065 USA. Please remember to include your complete name and address when contacting us.

ACCESSING THE DIGITAL ARCHIVE: Your monthly download notifications will have links to the different formats and to the digital archive. To access the digital archive at any time, log in at http://www.linuxjournal.com/digital.

LETTERS TO THE EDITOR: We welcome your letters and encourage you to submit them at http://www.linuxjournal.com/contact or mail them to *Linux Journal*, 9597 Jones Rd #331, Houston, TX 77065 USA. Letters may be edited for space and clarity.

SPONSORSHIP: We take digital privacy and digital responsibility seriously. We've wiped off all old advertising from *Linux Journal* and are starting with a clean slate. Ads we feature will no longer be of the spying kind you find on most sites, generally called "adtech". The one form of advertising we have brought back is sponsorship. That's where advertisers support *Linux Journal* because they like what we do and want to reach our readers in general. At their best, ads in a publication and on a site like *Linux Journal* provide useful information as well as financial support. There is symbiosis there. For further information, email: sponsorship@linuxjournal.com or call +1-281-944-5188.

WRITING FOR US: We always are looking for contributed articles, tutorials and real-world stories for the magazine. An author's guide, a list of topics and due dates can be found online: http://www.linuxjournal.com/author.

NEWSLETTERS: Receive late-breaking news, technical tips and tricks, an inside look at upcoming issues and links to in-depth stories featured on http://www.linuxjournal.com. Subscribe for free today: http://www.linuxjournal.com/enewsletters.

LINUX JOURNAL

EDITOR IN CHIEF: Doc Searls, doc@linuxjournal.com

EXECUTIVE EDITOR: Jill Franklin, jill@linuxjournal.com

TECH EDITOR: Kyle Rankin, lj@greenfly.net

ASSOCIATE EDITOR: Shawn Powers, shawn@linuxjournal.com

EDITOR AT LARGE: Petros Koutoupis, petros@linux.com

CONTRIBUTING EDITOR: Zack Brown, zacharyb@gmail.com

SENIOR COLUMNIST: Reuven Lerner, reuven@lerner.co.il

SENIOR COLUMNIST: Dave Taylor, taylor@linuxjournal.com

PUBLISHER: Carlie Fairchild, publisher@linuxjournal.com

ASSOCIATE PUBLISHER: Mark Irgang, mark@linuxjournal.com

DIRECTOR OF DIGITAL EXPERIENCE:
Katherine Druckman, webmistress@linuxjournal.com

GRAPHIC DESIGNER: Garrick Antikajian, garrick@linuxjournal.com

COVER DESIGN: Carty Sewill

ACCOUNTANT: Candy Beauchamp, acct@linuxjournal.com

Linux Journal is published by, and is a registered trade name of,
Linux Journal, LLC. 4643 S. Ulster St. Ste 1120 Denver, CO 80237

SUBSCRIPTIONS
E-MAIL: subs@inuxjournal.com
URL: www.linuxjournal.com/subscribe
Mail: 9597 Jones Rd, #331, Houston, TX 77065

SPONSORSHIPS
E-MAIL: sponsorship@linuxjournal.com
Contact: Publisher Carlie Fairchild
Phone: +1-281-944-5188

Engineers vs. Re-engineering

Doc Searls is a veteran journalist, author and part-time academic who spent more than two decades elsewhere on the *Linux Journal* masthead before becoming Editor in Chief when the magazine was reborn in January 2018. His two books are *The Cluetrain Manifesto*, which he co-wrote for Basic Books in 2000 and updated in 2010, and *The Intention Economy: When Customers Take Charge*, which he wrote for Harvard Business Review Press in 2012. On the academic front, Doc runs ProjectVRM, hosted at Harvard's Berkman Klein Center for Internet and Society, where he served as a fellow from 2006–2010. He was also a visiting scholar at NYU's graduate school of journalism from 2012–2014, and he has been a fellow at UC Santa Barbara's Center for Information Technology and Society since 2006, studying the internet as a form of infrastructure.

In an age when people are being re-engineered into farm animals for AI ranchers, it's the job of engineers to save humanity through true personal agency.

By Doc Searls

A few months ago, I was driving through Los Angeles when the Waze app on my phone told me to take the Stadium Way exit off the 110 freeway. About five other cars peeled off with me, and we became a caravan, snaking through side streets and back onto the freeway a few miles later. I knew Waze had to be in charge of us, since Waze is the navigation app of choice in Los Angeles, and it was beyond coincidence that all these cars took the same wild maze run through streets only locals knew well.

What was Waze up to here, besides offering its users (or a subset of them) a way around a jam? Was it optimizing traffic by taking some cars off the highway and leaving others on? Running an experiment only some AI understood? There was no way to tell. I doubt anyone at Waze could say exactly what was going on either. Algorithms are like that. So are the large and constantly changing data sets informing algorithms most of us with mobile devices depend on every day.

In *Re-engineering Humanity*, Brett Frischmann and Evan Selinger have dug deeply into what's going on behind the "cheap bliss" in our fully connected world.

What they say is that we are all subjects of *techno-social engineering*. In other words, our algorithmic conveniences are re-making us, much as the technologies and techniques of agriculture re-makes farm animals. And, as with farming, there's an extraction business behind a lot of it.

They say "humanity's techno-social dilemma" is that "companies, institutions, and designers regularly treat us as *programmable objects* through personalized technologies that are attuned to our personal histories, present behavior and feelings, and predicted futures."

And we are not innocent of complicity in this. "We outsource memory, decision-making and even our interpersonal relations...we rely on the techno-social engineers' tools to train ourselves, and in doing so, let ourselves be trained."

There are obvious benefits to "delegating physical, cognitive, emotional and ethical labor to a third party", such as Waze, but there are downsides, which Brett and Evan number: 1) passivity, 2) decreased agency, 3) decreased responsibility, 4) increased ignorance, 5) detachment and 6) decreased independence. On the road to these diminished human states, we have "fetishised computers and idealized computation".

Doing both means "we work on problems best solved by computation", which in turn leads to "the imperialism of instrumental reason and the improper assumption that all problems are comprehensible in the language of computation and thus can be solved with the same set of social and technological tools".

They see today's faith in computational technology as the latest expression of belief in Frederick Taylor's theory of scientific management. More than a theory, Taylorism gave us norms for producing mass-market goods that have been with us for more than a century and have hardly gone away. In *The Cluetrain Manifesto* (which four of us wrote in 1999), Chris Locke explains:

> Taylor's time-and-motion metrics sought to bring regularity and predictability to bear on the increasingly detailed division of labor. Under such a regimen, previously holistic craft expertise rapidly degraded into the mindless execution of single repetitive tasks, with each worker performing only one operation in the overall process. Because of its effect on workers' knowledge, de-skilling is a term strongly associated with mass production. And as skill disappeared, so did the unique voice of the craftsman. The organization was elegantly simple, if not terribly humane.

"At one level", write Brett and Evan, "Taylor's scientific management system is a type of data-dependent technology. Taylorism is one of the best early examples of data-driven innovation, a concept currently in vogue. Taylor's systems included the techniques for both gathering data and putting such data to use in managing people. Taylor's system thus encompassed the surveillance techniques employed by the 'efficiency experts'" of Taylor's time—and of ours.

Surveillance of people is now the norm for nearly every website and app that harvests personal data for use by machines. Privacy, as we've understood it in the physical world since the invention of the loincloth and the door latch, doesn't yet exist. Instead, all we have are the "privacy policies" of corporate entities participating in the data extraction marketplace, plus terms and conditions they compel us to sign, either of which they can change on a whim. Most of the time our only choice is to deny ourselves the convenience of these companies' services or live our lives offline.

Worse is that these are proffered on the Taylorist model, meaning mass-produced.

In "Contracts of Adhesion—Some Thoughts about Freedom of Contract" (*Columbia Law Review*, July 1943), Friedrich Kessler explained how these shitty non-agreements came to be:

> The development of large scale enterprise with its mass production and mass distribution made a new type of contract inevitable—the standardized mass contract. A standardized contract, once its contents have been formulated by a business firm, is used in every bargain dealing with the same product or service. The individuality of the parties which so frequently gave color to the old type of contract has disappeared. The stereotyped contract of today reflects the impersonality of the market....Once the usefulness of these contracts was discovered and perfected in the transportation, insurance, and banking business, their use spread into all other fields of large scale enterprise.

Rather than obsolesce these kinds of contracts, digital technology and the internet allowed companies to automate them. The effects are not good for the human side of each contract. Brett and Evan explain, "Our current online contracting regime is a compelling example of how our legal rules coupled with a specific technological environment can lead us to behave like simple stimulus-response machines—perfectly rational, but also perfectly predictable and ultimately programmable."

In a chapter titled "On Extending Minds and Mind Control", Brett and Evan visit the ways humans are diminished as well as enlarged when programmed by their digital conveniences. To start getting what they mean, it helps to remember that every personal technology we operate—shoes, bikes, pens, books, cars, hammers,

airplanes—extend our senses and enlarge our agency. It is no mistake that drivers speak in first-person possessive pronouns about the parts of the cars they operate: *my* tires, *my* engine, *my* fenders. To drive a car is to become one.

But what happens when our senses extend beyond the metal carapace we wear when we drive a car—outward through the unseen systems guiding our selves and every other car on the road? In this state we are not GPS satellites and Google data centers, but rather puppets at the ends of digital strings pulled by AI puppeteers.

We surely appreciate and rely on what they provide us, but we also yield agency in the process of blurring between our automotive selves and a vast system of dependencies, which even if they are doing good things for us, make us less than human—or, in Brett and Evans' words, "simple machines under the control and influence of those in control of technologies". Inevitably, they also say, "traffic engineers will assume the role of social planners." But the traffic engineers they're talking about are not the human kind working for highway departments, but machines run by companies making navigation apps, all of which have purposes beyond providing personal and civic goods.

The challenge Brett and Evan pose in their concluding chapter is "how to sustain the freedom to be off, to be free from techno-social engineering, to live and develop within undetermined techno-social environment". Toward these they set a buffet of possible approaches, three of which can be addressed by the agency of *Linux Journal* readers, many of whom are engineers by trade:

1. Challenge conventional wisdom, ideas and theories that perpetuate existing logics and engineer complacency.

2. Create gaps and seams between smart techno-social systems that constrain techno-social engineering and techno-social engineering creep.

3. Engineer transaction costs and inefficiencies to support human flourishing through the exercise and development of human capabilities.

Then they give a nod to decentralization and go into possible reforms to

legal systems.

In the book's last two pages, they also give a nod toward my own work: "Doc Searls and his colleagues at Customer Commons have been working for years on standardized terms for customers to use in managing their relationships with websites and other vendors", noting that the General Data Protection Regulation (GDPR) in Europe at least makes it possible "to imagine alternative contracting practices and more nuanced contractual relationships".

In the book's final paragraph, they say "Doc Searls' dream of customers systematically using contract and related tools to manage their relationships with vendors now seems feasible. It could be an important first step toward flipping the scientific-management-of-consumers script we've become so accustomed to."

The engineers among us have a hard job. But it should help to know how high the stakes are, and how we're already embedded so deeply in a re-engineered dystopia that we can't see how tragically ironic cheap bliss really is. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

LETTERS

On Project Fi

I just finished reading Shawn Powers' "A Look at Google's Project Fi" from the July 2018 issue, and I wanted to chime in from the bus I am currently riding from Siem Reap to Phnom Penh, Cambodia, using my Pixel on Project Fi. One of the greatest benefits for me is the ability to use it in almost every country in the world. I have personally used Project Fi in Japan, Mexico, Singapore, Malaysia, Korea, Cambodia, Singapore and of course the US. I keep a data-only SIM in a spare (non-fi-compatible) phone and use that one via USB tether to a travel router to provide access to several devices when no WiFi is available.

When I can connect to a hotel's WiFi, Google Fi lets me make calls back to the US at no charge.

One more thing: if you set up a Google Voice number, you can use the Hangouts Dialer to make and receive calls from a data-only SIM, and SMS comes through Hangouts. For that to work, you need to use a different account from the one that Fi is set up on.

Regards, and keep up the great work!

—J. Jordan

Shawn Powers replies: I haven't been able to test the international usefulness myself (apart from briefly picking up a Canadian tower when I was in Michigan's upper peninsula). It's great to hear how well it works! Also, the Hangouts tip is great. I could see that being an awesome solution for a family trying to save money with younger kids. That way they still could make calls, but not need a new "line" to do it. Thanks!

More on Project Fi

I liked Shawn Powers' article about Project Fi, but with the restriction to T-Mobile for data only, you also should look into the Ting service as an alternative. It's $6 per phone for the "line" to keep the SIM active, and then everything else is shared and pay-as-you-go. It works on virtually any GSM phone. You can buy the SIM cards at

Kroger and so on.

I have a cheap ($60 new) Blu Android 6.0 smartphone that I use with Ting. I use it only for sending text messages from our home security system. The basic $3 for 300 texts means my monthly bill is around $11.27 after taxes, or only $8.27 if no alert texts get sent. This is very much worthwhile for the peace of mind that comes from knowing I'll get alerts even if the internet is down, along with alerts when the AC power goes down and is restored.

Since I was already using MQTT in my home security system, it was easy to install Termux and Termux-API from the play store, node-red using apt-get and then in a web browser create a flow (what node-red calls a program) to accept SMS messages over MQTT and send them out on the Ting/T-Mobile network using Termux-API.

If you can find something cheaper, it'd be fantastic, but I've been looking for a long time. The free FredoomPop service actually meets most of my smartphone needs (100MB/month free data on AT&T, 400MB/month free on Sprint), but they use a "special" VOIP dialer and messaging app that doesn't properly register its "Android Intent", so the termux-node-red SMS messages end up in the default Android Messages App and never get sent.

I'm very happy that *LJ* has made a comeback!

If you want an idea for an article, take a look at this project I've put up on GitHub: https://github.com/wb666greene/SecurityDVR_AI_addon.

The basic idea is to use a Raspberry Pi3 and Movidius Neural Compute Stick to add AI "person detection" to snapshot images from an existing video security DVR and send "push" notifications via SMS and email with an image when a person is detected when and where they shouldn't be. The Movidius can do about ten images per second. My system has been running for over a month now monitoring eight cameras, and the false alarm rate is effectively zero—the only bogus alerts were when my wife put some hand-washed clothes or a chair on the pool deck, and another was when she put her

floppy sun hat on top of the tower fan on the patio—both kind of looked like a person at a quick glance.

I'm also about to put up a project doing a standalone AI-enhanced security camera using a Raspberry Pi (2B or above) and Pi NoIR camera module. It achieves only about one frame every two seconds, but the highly reliable push alerts make it well worth the ~$80 it takes for parts. I've uncovered what seems to be a bug in the openCV dnn module. I'm testing a workaround on two systems as I type this, and if they run for a week or more, I'll put the project up, but I'd prefer to find a real solution to the issue first. I've got questions out, but so far no responses.

—Wally

Shawn Powers replies: Thanks for the tip on Ting. I'll offer another possibility, one that I used last year for our foreign-exchange student: RedPocket. It looks like the price isn't as low as it was when I bought it, but I got an annual contract, with SIM, which had 500MB of data, 500 minutes of voice and 500 text messages per month for $60. It looks like the same deal is $99 now, but maybe it goes on sale. Nevertheless, pre-paid plans are amazing for certain circumstances, and it's nice to see affordable options out there!

As far as the project goes, I'd love to see details about it. Heck, I have a Raspberry Pi sitting on my desk looking for a project. I'll look at your GitHub page and might have some questions for you. I'll be sure to give you accolades if I set up the project!

Regarding Work the Shell, "Shuffling Letters and Words" in the July Issue

Something that works on Mac OSX (or pretty much anything with `$RANDOM` and bourne-like shell):

```
cat 1984.txt | sed -e '1i \

cat <<HERE' -e 's/^/\$RANDOM    /' -e '$a \
```

```
HERE' | sh | sort | cut -f2
```

Note: there's a tab after `$RANDOM`.

—Christopher Cox

Dave Taylor replies: Nice, streamlined solution, Christopher. Thanks for writing in and sharing it!

BirdCam—Image Capture and Classification

To Shawn Powers: I am a fan of *Linux Journal* and have come across something I think might pique your interest.

I found an article while searching for home automation and security ideas. A guy named Robin Cole created a system to classify images as bird or "not bird" at his home, using a few of your favourite things: birds, bird feeder, Raspberry Pi, Synlogy NAS and a webcam.

You might even be challenged to identify the bird in the image too, who knows.

Have a look here.

As far as I could see, most of it is open source too.

—Hanro

Shawn Powers replies: Whoa, that's awesome! One of the problems I have with my motion detection is false positives. If you watch my nightly archives, you'll see a lot of the "motion" is non-feather-related. I'll have to check it out, thanks for the link!

Question for Kyle Rankin Regarding Odroid Xu4 NAS

Can you please detail the software stack, configuration, and back-up scripts that

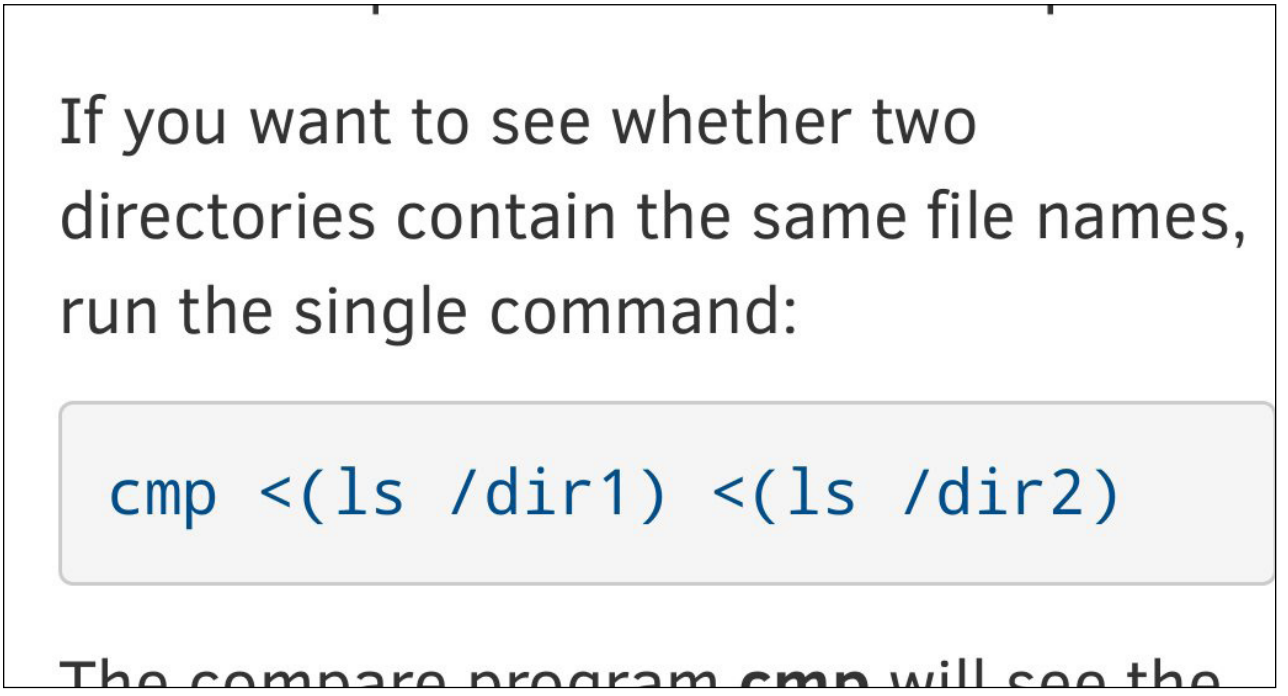
power your Odroid Xu4/Mediasonic Probox NAS solution? Many thanks,

—Lloyd

Kyle Rankin replies: Thanks for the letter. It's true that I mostly focused on the hardware side of my NAS in my "Papa's Got a Brand New NAS" article. I should do a follow-up column at some point in the future about the software side. In summary, I use an off-the-shelf Debian image and standard NFS and CIFS file-sharing services to share directories with other computers on my network.

From Social Media

Davide Principi @davideprincipi: Old but gold: '97 @linuxjournal article about named pipes: https://www.linuxjournal.com/article/2156!



Brad Beyenhof @augmentedfourth: @reuvenmlerner I enjoyed your article on Dataclasses in @linuxjournal and I'm interested in your courses. Is there a difference in level/content between "Practice Makes Python" and the Weekly Python Exercise?

Reuven M. Lerner @reuvenmlerner: Replying to @augmentedfourth @linuxjournal Glad you enjoyed it! Practice Makes Python is a course with exercises. You do it on your own, either with or without video instructions/explanations. It's aimed at people who have a more basic background, roughly people who have been through my intro course.

Reuven M. Lerner @reuvenmlerner: Replying to @reuvenmlerner @linuxjournal @augmentedfourth Let me know if you have further questions! You can learn more about WPE at http://WeeklyPythonExercise.com or my courses in general at http://store.lerner.co.il.

Canonical Ltd @Canonical Embracing Snaps: an Interview with @Canonical and @SlackHQ http://bit.ly/2tPNxUH via @linuxjournal

John Nordberg @dgm9704: Replying to @Canonical @linuxjournal @SlackHQ I'm wondering, is it snaps like snapping your fingers, snaps like snapping into place, or snaps like schnapps? ;-)

SEND *LJ* A LETTER *We'd love to hear your feedback on the magazine and specific articles. Please write us here or send email to ljeditor@linuxjournal.com.*

PHOTOS *Send your Linux-related photos to ljeditor@linuxjournal.com, and we'll publish the best ones here.*

Patreon and *Linux Journal*

PATREON

Together with the help of *Linux Journal* supporters and subscribers, we can offer trusted reporting for the world of open-source today, tomorrow and in the future. To our subscribers, old and new, we sincerely thank you for your continued support. In addition to magazine subscriptions, we are now receiving support from readers via Patreon on our website. *LJ* community members who pledge $20 per month or more will be featured each month in the magazine. A very special thank you this month goes to:

- Appahost.com
- David Breakey
- Dr. Stuart Makowski
- Josh Simmons
- Magnus Magicman
- Mostly_Linux
- NDCHost.com
- Robert J. Hansen
- Chris Short

BECOME A PATRON

FOSS Project Spotlight: SIT (Serverless Information Tracker)

In the past decade or so, we've learned to equate the ability to collaborate with the need to be online. The advent of SaaS clearly marked the departure from a decentralized collaboration model to a heavily centralized one. While on the surface this is a very convenient delivery model, it simply doesn't fit a number of scenarios well.

As somebody once said, "you can't FTP to Mars", but we don't need to go as far. There are plenty of use cases here on Earth that are less than perfectly suited for this "online world". Lower power chips and sensors, vessel/offshore collaboration, disaster recovery, remote areas, sporadically reshaping groups—all these make use of central online services a challenge.

Another challenge with centralization is somewhat less thought of—building software that can handle a lot of concurrent users and that stores and processes a lot of information and never goes down is challenging and expensive, and we, as consumers, pay dearly for that effort.

And not least important, software in the cloud removes our ability to adapt it perfectly for use cases beyond its owner's vision, scope and profitability considerations. Convenience isn't free, and this goes way beyond the price tag.

SIT is a free, open-source project that addresses these and other concerns in software that enables us to collaborate. It allows sporadically connected parties to continue collaborating seamlessly, over just about any digital transport (ranging from a P2P network to a USB drive). At its core, it's a very small tool that records every change as an immutable, additive-only set of files and allows this information to be displayed and

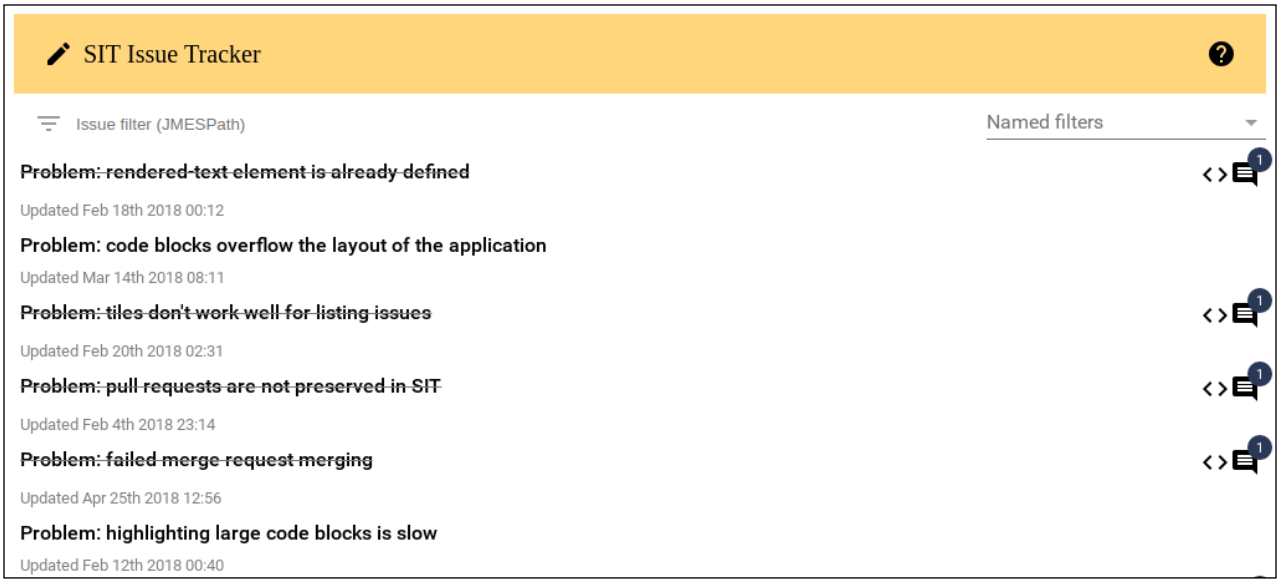


Figure 1. SIT Issue Tracker

operated on in a familiar way, though browser-based applications or the command line.

Although its foundation is rather generic, its first real application is in issue tracking, and it enables a lot of scenarios that were previously rather difficult to achieve. For example, if a SIT repository is committed to a project repository, this allows you to see a snapshot of all issues for any revision, making it much easier to maintain separate versions or trace changes. Another interesting feature is its merge request functionality, where a patch, by its nature, can contain file changes that affect a project's issues, giving enormous flexibility in managing dependent issues (say you developed a feature and want to attach a "to-do" list to it as a part of the patch, so those new issues will appear only once the patch has been merged—with SIT this is a rather trivial task).

SIT is still a young project, and there are still a lot of cases to figure out. It's quite interesting to see how so many assumptions are being challenged once we remove the central party from the equation.

Yurii Rashkovskii is an open-source advocate, software developer and instigator on a mission to make software cheaper to produce and maintain. He's been involved in different communities and projects, starting some and contributing to others, including projects as Elixir, NixOS and ecosystems of others. He can be occassionally seen at software conferences, meetups or on the road, cycling. He lives in Vancouver, Canada, and spends a lot of time in Asia.

Tech Tip: Easy SSH Automation

A script a day will allow you some freedom to play and build other useful and more complicated scripts. Every day, I attempt to make my life easier—by this I mean, trying to stop doing the repetitive tasks. If a process is repeatable; it can be scripted and automated. The idea to automate everything is not new, but try automating a command on a remote host.

SSH is very flexible, and it comes with many options. My absolute favorite is its ability to let you run a command on a remote server by passing the `-t` flag. An example:

```
ssh -t adam@webserver1.test.com 'cat /etc/hosts'
```

This will `ssh` to webserver1.test.com, then run `cat /etc/hosts` in your shell and return the output.

For efficiency, you could create an ssh key pair. It's a simple process to create a passwordless public and a private key pair. To set this up, use `ssh-keygen`, and accept the defaults ensuring you leave the password blank:

```
ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/adam/.ssh/id_rsa): y
Enter passphrase (empty for no passphrase): LEAVE BLANK
Enter same passphrase again:
Your identification has been saved in /home/nynet/.ssh/id_rsa.
Your public key has been saved in /home/nynet/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:jUxrQRObADE8ardXMT9UaoAcOcQPBEKGU622646P8ho
```

```
 ↪adam@webserver1.test.com
The key's randomart image is:
+---[RSA 2048]----+
|B*++*Bo.=o       |
|.+.              |
|=*=              |
+----[SHA256]-----+
```

Once completed, copy the public key to the target server. To do this, use `ssh-copy-id`:

```
ssh-copy-id adam@webserver1.test.com
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed:
"/home/adam/.ssh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s),
 ↪to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if
 ↪you are prompted now it is to install the new keys
adam@webserver1.test.com's password: ********
Number of key(s) added:        1
```

You will be asked for the password of the target server.

If you have set this up correctly, you won't be asked for your password next time you `ssh` to your target.

Execute the original example. It should be quicker now that you don't need to enter your password.

If you have a handful of servers and want to report the running kernel versions, you can run `uname -r` from the command line, but to do this on multiple devices, you'll need a script.

Start with a file with a list of your servers, called server.txt, and then run your script to iterate over each server and return the required information:

```
#!/bin/bash
if [ -f server.txt ]; then
        for server in $(cat server.txt); do
                ssh -t adam@$server  '
                echo $(uname -r)                '
        done
else
        echo 'No server.txt file'
fi
```

The `if` statement is checking to ensure that there's a file called server.txt. The `for` loop creates the variable called `server` for each target in server.txt, it then connects and fetches the kernel information.

In conclusion, with slight modifications, you can have an army of scripts to run in cron or manually; these scripts will become your toolbox to freedom.

Adam McPartlan is Father of Twins - Linux lover, Open Source Enthusiast - LFCS, AWS Cloud Practitioner. Follow him on Twitter: @mcparty.

If you have a short tech tip you'd like to share with *Linux Journal* readers, send an email with Tech Tip in the subject line to ljeditor@linuxjournal.com.

Copy and Paste in Screen

Put the mouse down, and copy and paste inside a terminal with your keyboard using Screen.

Screen is a command-line tool that lets you set up multiple terminal windows within it, detach them and reattach them later, all without any graphical interface. This program has existed since before I started using Linux, but first I clearly need to address the fact that I'm even using Screen at all prior to writing a tech tip about it. I can already hear you ask, "Why not tmux?" Well, because every time someone tries to convince me to make the switch, it's usually for one of the following reasons:

- Screen isn't getting updates: I've been happy with the current Screen feature set for more than a decade, so as long as distributions continue to package it, I don't feel like I need any newer version.

- tmux key bindings are so much simpler: I climbed the Screen learning curve more than a decade ago, so to me, the Screen key bindings are second nature.

- But you can do vertical and horizontal splits in tmux: you can do them in Screen too, and since I climbed that learning curve ages ago, navigating splits are part of my muscle memory just like inside vim.

So now that those arguments are out of the way, I thought those of you still using Screen might find it useful to learn how to do copy and paste within Screen itself. Although it's true that you typically can use your mouse to highlight text and paste it, if you are a fan of staying on the home row like I am, you realize how much faster and more efficient it is if you can copy and paste from within Screen itself using the keyboard. In fact, I found that once I learned this method, I ended up using it multiple times every day.

Enter Copy Mode

The first step is to enter copy mode from within Screen. Press Ctrl-a-[to enter copy mode. Once you're in this mode, you can use arrow keys or vi-style keybindings to navigate up and down your terminal window. This is handy if you are viewing a log or other data that has scrolled off the screen and you want to see it. Typically people who are familiar with copy mode just use it for scrolling and then press q to exit that mode, but once you are in copy mode, you also can move the cursor to an area you want to copy.

Copy Text

To copy text once in copy mode, move your cursor to where you want to start to copy and then press the space bar. This will start the text selection, and you'll see the cursor change so that it highlights the text as you then move the cursor to select everything you want to copy. Once you are done selecting text, press the space bar again, and it will be copied to Screen's copy buffer. Once text is copied to Screen's clipboard, it automatically will exit copy mode.

Paste Text

Once you have text in the copy buffer, you can switch to a different Screen window, open a text document in your terminal or otherwise move the cursor to where you want to paste. When you are ready to paste, press Ctrl-a] (note that this is the opposite bracket from what you use in copy mode) to paste. As with other clipboards, you can paste multiple times as needed.

Conclusion

For those of you like me who are still using Screen, hopefully this quick copy-and-paste tip will improve your productivity so you aren't tempted by those cool kids using tmux. As I mentioned before, once I committed the copy-and-paste method to muscle memory, I use it constantly without even thinking about it, much like you would with a mouse.

—Kyle Rankin

Telecommuting Tips

With all the collaboration technology available for offices today, there's no reason telecommuters can't be as productive and as connected as other team members.

I live in the San Francisco Bay Area, known for high-tech companies, horrible traffic and high cost of living. When it came time for me to buy a house, I chose an area that left me with a 90–120-minute commute, depending on traffic and the time of day, so through the years, I've negotiated work-from-home days and have experience with telecommuting at companies of various sizes with different proportions of remote workers. Telecommuting is not only more convenient for many employees, it also can get the best work out of people, because it can grant better opportunities to focus and lets employees get right to work instead of spending hours getting to and from work. Unfortunately, many places inadvertently sabotage their telecommuters with bad practices, so here are a few tips to help make telecommuting successful.

Invest in Good Teleconference Hardware

I've attended many video conferences where the audio was so horrible, I might as well have not joined. Or worse, there was a time when one speaker was loud and clear, but when the conversation went to the other side of the table, it was inaudible. Although it's nice to have quality cameras, having quality microphones is critical. Make sure each of your meeting rooms has quality microphones that can pick up sounds all around the meeting table, and make sure attendees speak up. Relying on the microphone on someone's laptop just doesn't cut it for meetings involving more than two people. Although it's considered good meeting etiquette to have only one person speak at a time, this protocol is extra important if you have anyone calling in, as cross-talk makes it all but impossible to hear either conversation even over a good microphone.

Add Video Conference Links to Every Meeting

Make it a habit to add a link to your video conference room for each meeting you

create, even if all of the attendees are expected to be in the office. This habit ensures that when you realize you forgot to invite a remote workers, you aren't scrambling to figure out how to set up the video conference, plus sometimes even team members in the office need to work from home at the last minute. If your scheduling software can do this automatically, even better (some do this by having each meeting room in a contact list and inviting the relevant meeting room to the meeting). Also make sure you set this up for all-hands company-wide meetings.

Remember to Invite Remote Workers to Impromptu Meetings

Companies that rely on a central office often treat telecommuters as second-class citizens. Not only are they left out of impromptu hallway conversations, when those hallways move to a meeting room, the attendees also often forget to invite any remote workers who may be stakeholders in the decision and find out about major decisions only after the fact. Try to keep relevant team discussions in your work chat, so remote workers can participate (this has the added advantage of creating a nice log of your conversation, plus multiple people can participate at once). If you do end up moving a discussion into a meeting room, remember to invite your remote team members.

Take Advantage of Group Chat, Even in the Office

There are many advantages to using group chat for team communication, even if team members are in the same office. The virtual "tap on the shoulder" in a group chat is much less disruptive to someone's focus than the actual in-person tap on the shoulder, which means if someone is deeply focused on a task, your chat notification may not disrupt them, although physically standing behind them forces them to stop anything they are doing. By sticking to group chat for these kinds of communications, you can be sure that remote members of the team aren't left out of any discussions, and every person feels like an equal member of the team.

Be Responsive in Chat

If you work remotely and a lot of your team is in an office, it's more important than ever to stay responsive in chat. In an office, people know you are there by whether you are at your desk, but when telecommuting, it all comes down to whether you

respond when someone contacts you in chat. If your chat program provides ways to escalate notifications from your desktop to the phone app, take advantage of them. That way if you step away from your desk for a moment, your phone still can tell you when someone needs you. If you are going to step away from your desk for an extended amount of time to get lunch or run an errand, make sure someone on your team knows (or set a proper away message if your chat supports it). If you are in the office, realize that chat is the main way remote workers will communicate, so try to reward their responsiveness by being responsive yourself.

In summary, the key to telecommuting being a success is to treat remote team members as equals and to take advantage of all of the great collaboration tools that exist these days to keep teams connected. These small steps can make all the difference in helping remote workers be productive and feel like part of the team.

—Kyle Rankin

Astronomy on KDE

I recently switched to KDE and Plasma as my main desktop environment, so I thought I'd start digging into some of the scientific software available on KDE. First up is KStars, the desktop astronomy program.

KStars probably won't be installed with the standard KDE desktop, so you may need to install it. If you're using a Debian-based distribution, you can install KStars with the following command:

```
sudo apt-get install kstars
```

When you first start it, KStars asks for your current location, and then it gives you the option of installing several extra information files to add to the list of objects that KStars knows about and can display. Once those steps are finished, KStars begins with the current sky at the location you entered earlier.

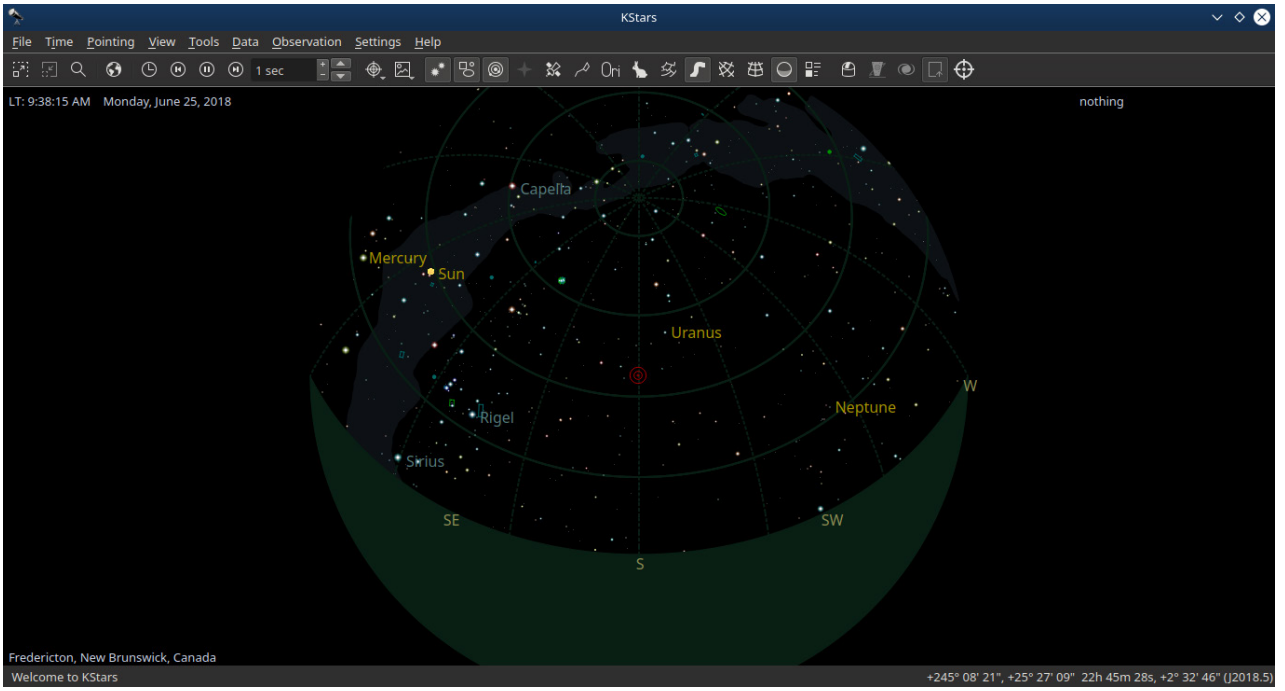


Figure 1. On startup, KStars shows you the current layout of the sky in your location.

So, what can you do with KStars? If you've used programs like Stellarium before, you'll find that you can do the same types of tasks with KStars. You can use your mouse to click and drag the display to change the direction you're facing. The cardinal directions are labeled along the outside of the circle of the sky, and you can zoom in and out to change the field of view. If you see an object you want to examine further, you can double-click it to center it on the display and tag it as the current object of interest.

Depending on what catalogs of data you installed, some of the objects may have more or less information available. For example, selecting the planet Uranus and zooming all the way in shows a reasonably detailed image of the planet, including the ring orientation.

Quite a few options are available for controlling what's shown in the main window. The toolbar across the top of the window allows you to toggle the following items: stars, deep sky objects, solar system objects, supernovae, satellites, constellation lines, constellation names, constellation art, constellation boundaries, Milky Way,

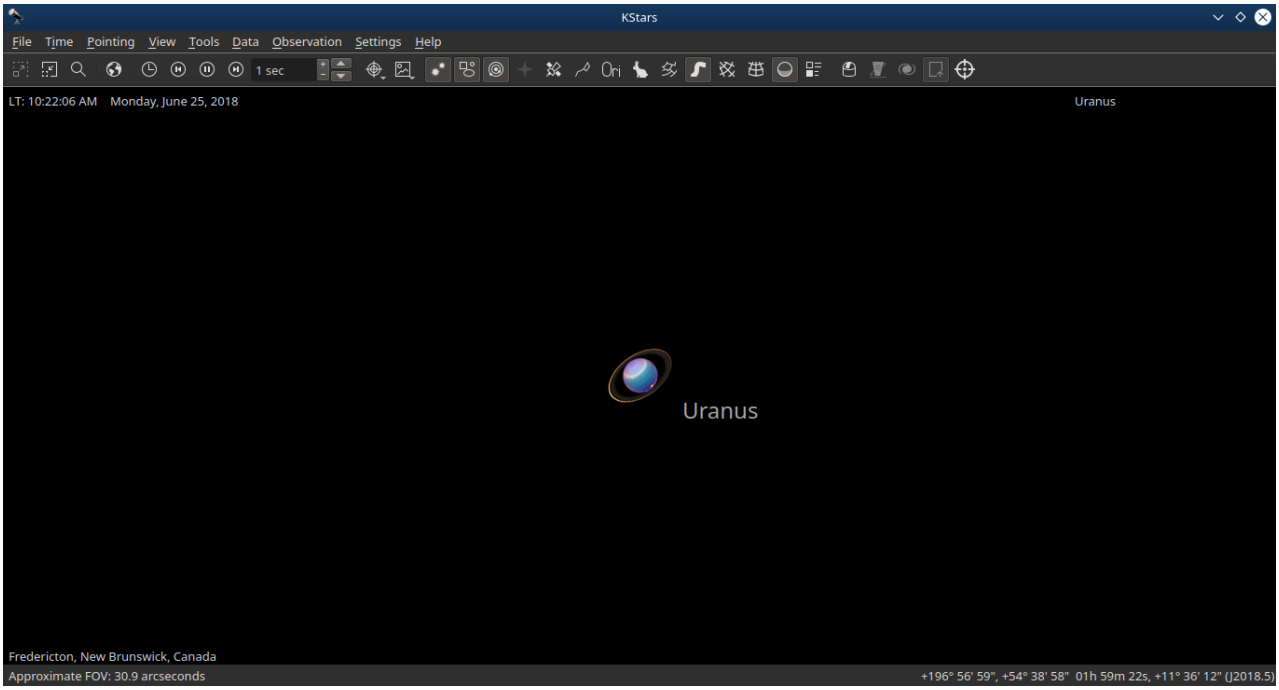


Figure 2. You can easily select and zoom in to objects of interest in KStars.

equatorial coordinate grid, horizontal coordinate grid and opaque ground. This allows you to customize the display so that it shows only what you're interested in at the time. The last display option is to toggle the "What's Interesting" pane.

You can click one an item in the pane, and then an object of that type will be highlighted in the display. The pane also changes to show a detailed list of all of those objects.

What if you want to change the current viewing conditions? Maybe you want to see what will be visible when you're on vacation next week? It's easy to change the time and location. Simply click the globe in the toolbar to pop up the location dialog.

If you click the clock icon in the toolbar, the time dialog pops up where you can set the viewing time and date, allowing you to see what's visible from anywhere and anytime. You also can change what information is available.

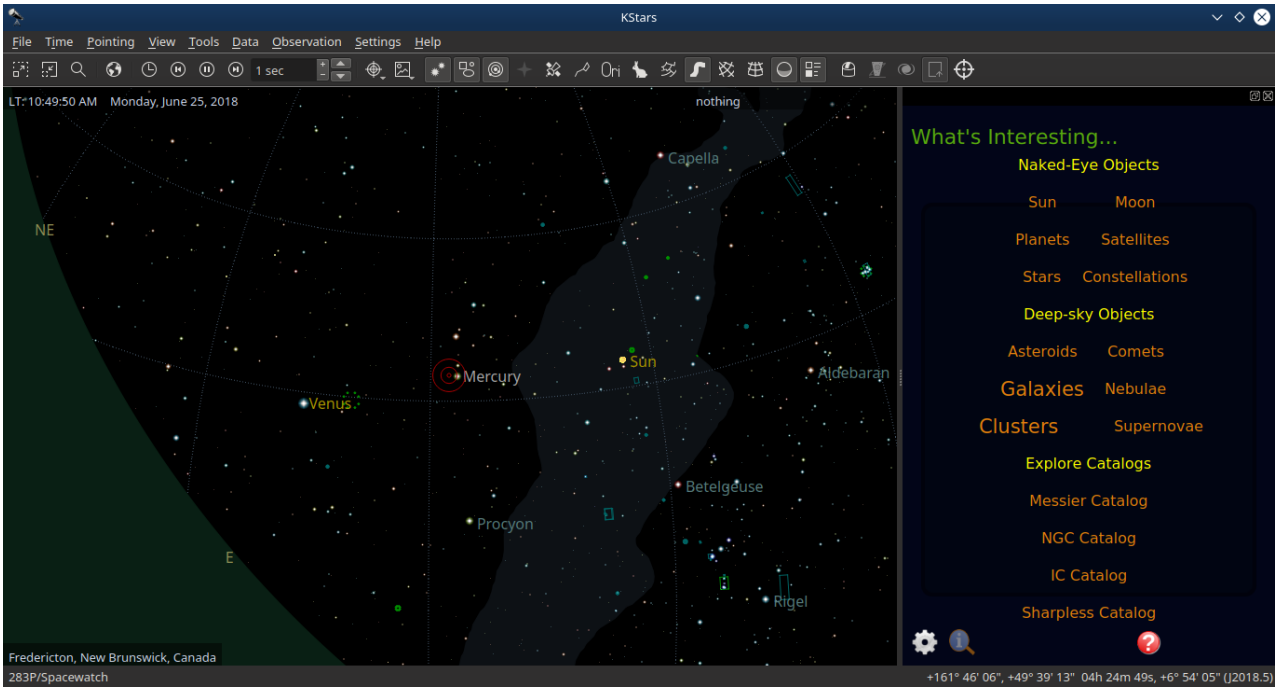


Figure 3. The "What's Interesting" pane lets you focus on a particular type of object visible in the sky with the current location and time.

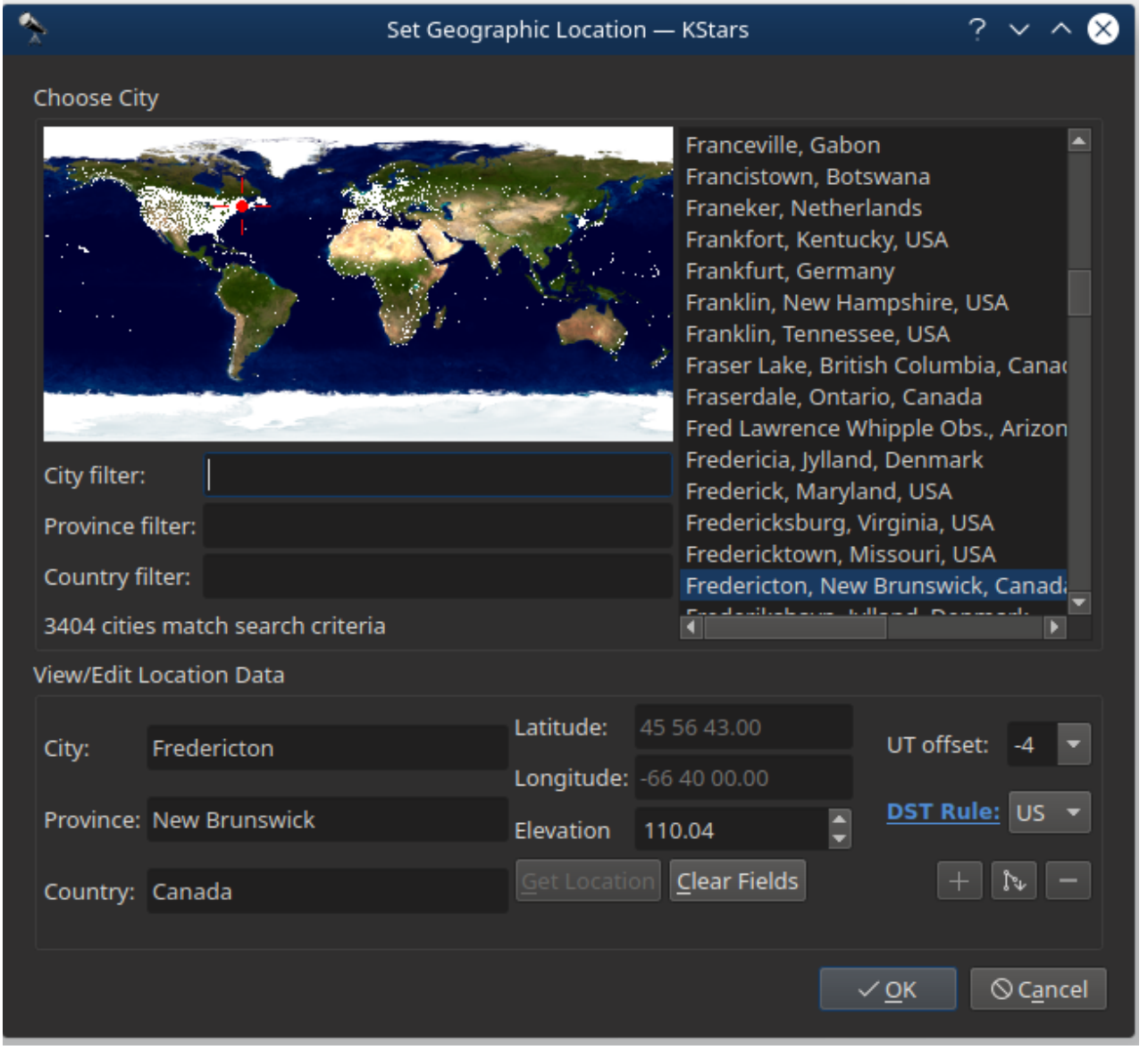


Figure 4. The location dialog lets you set where in the world you are, so you can see what's visible in the sky from that location.

If you declined to download extra information files when you first started KStars, you can go back in to that section and select some catalogs. Clicking Data→Download New Data opens a dialog window where you can select extra catalogs to install.

For objects that vary over time, you can download updated data by clicking

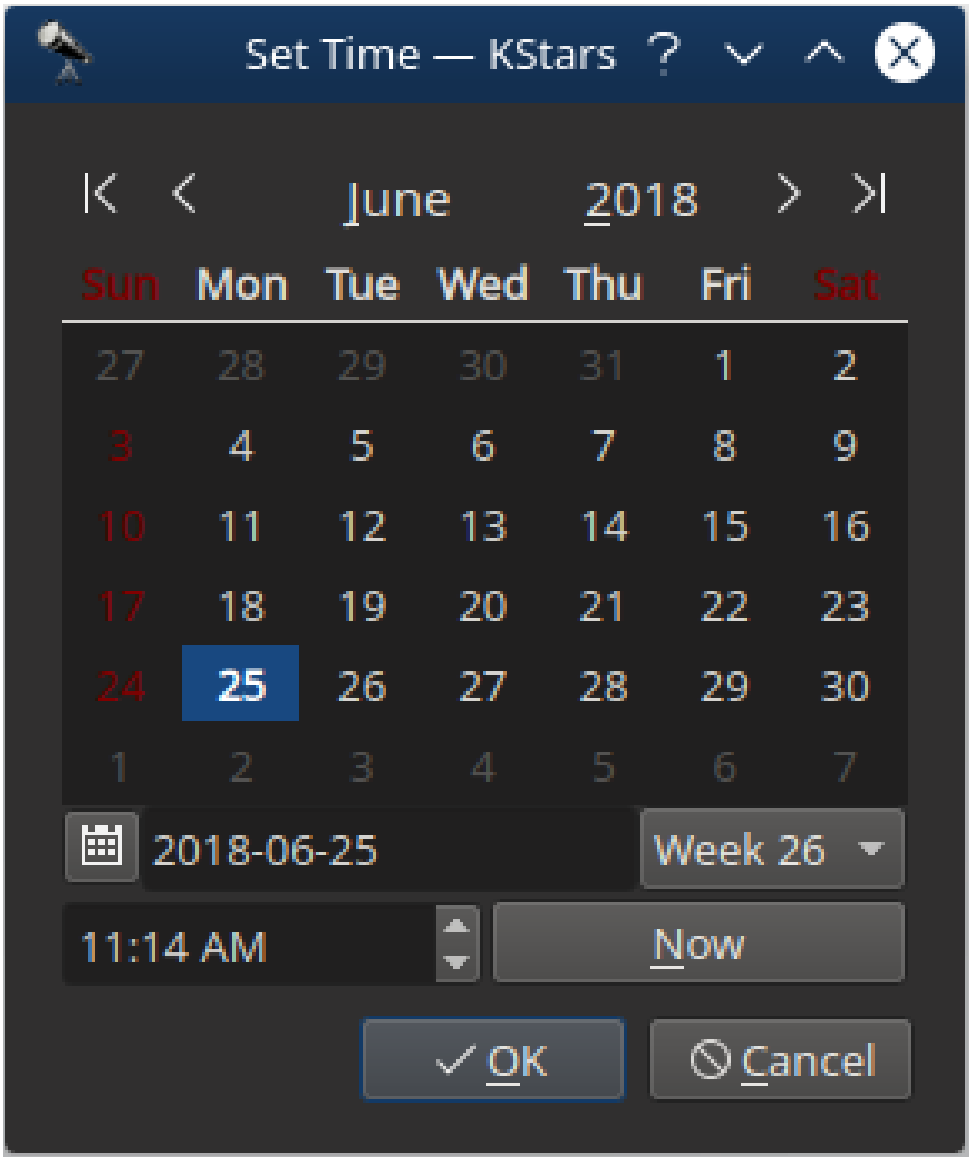


Figure 5. You can set the time and date to see what would be visible a given time and date with the time dialog.

Data→Updates, and selecting one of comets, asteroids, satellites or recent supernovae.

If your telescope supports computer control, you can use KStars to connect to it and give it instructions on what to do. Clicking Tools→Devices→Telescope Wizard opens a new window that walks you through the steps for adding your telescope's details to KStars.

You can manage these telescopes, along with a huge number of other potential devices, by clicking Tools→Devices→Device Manager, which opens a window of

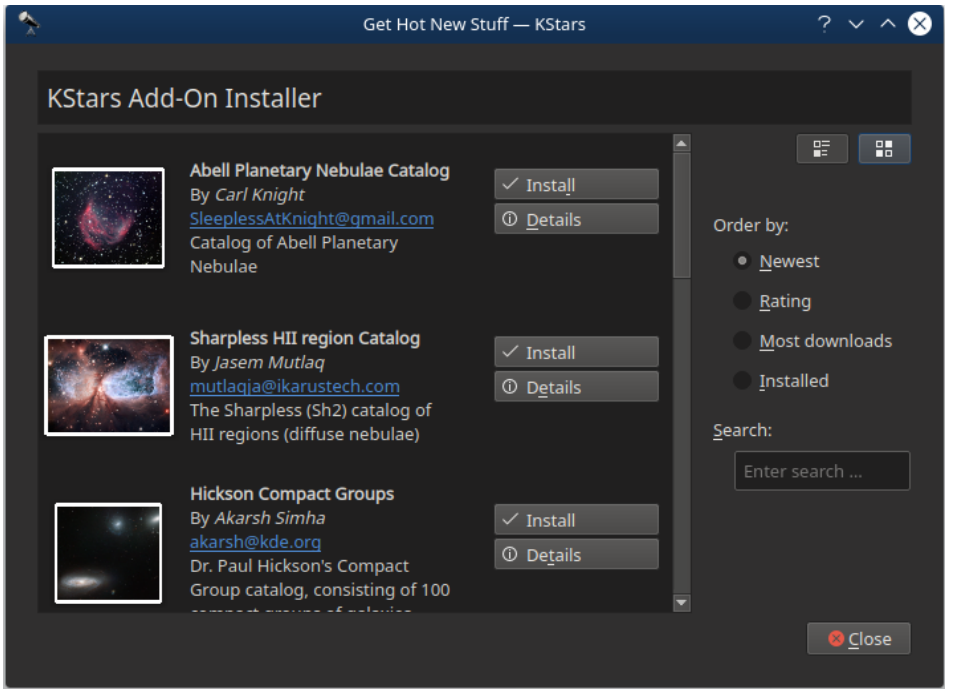


Figure 6. You can select from several extra data catalogs to add even more objects to KStars.

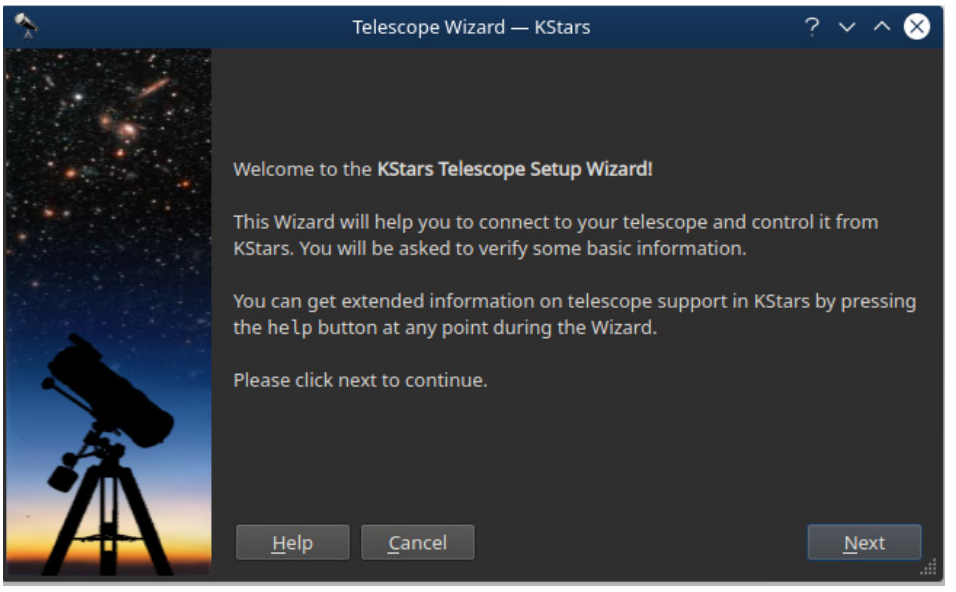


Figure 7. You can go through the telescope wizard to add the details for your particular device and allow KStars to communicate with it.

devices that KStars understands grouped by type.

Kstars also understands Ekos, the cross-platform observatory control and automation tool from the Open Astronomy Instrumentation group. The group also provides

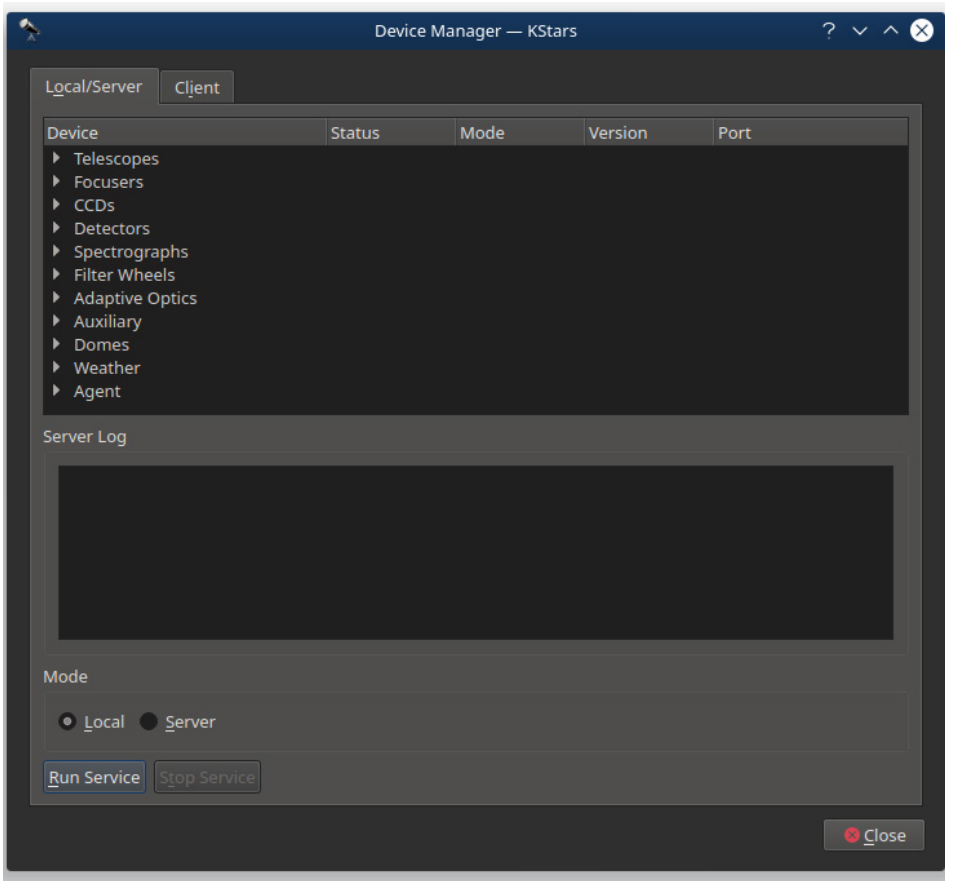


Figure 8. You can control a large number of potential devices with KStars.

the Instrument Neutral Distributed Interface (INDI) library, which KStars also understands. Clicking Tools→Ekos opens a new window where you can go through the steps to control your entire observatory. You can create a profile of which devices are going to be used, and then start the INDI library and connect to those devices. You then can start collecting data from the sensor devices you defined in the profile.

Although several options are available for doing astronomy from the desktop, if you're already using the KDE environment, you definitely should look at KStars. It has all of the usual features that you will need, and it's tightly integrated to the KDE environment of applications.

Visit https://edu.kde.org/kstars for more information.

—Joey Bernard

FOSS Project Spotlight: Run Remote Tasks on Linux and Windows with Puppet Bolt

Puppet, the company that makes automation software for managing systems and delivering software, has introduced Puppet Bolt, an open-source, agentless multiplatform tool for running commands, scripts, tasks and orchestrated workflows on remote Linux and Windows systems.

The tool, which is freely available as a Linux package, Ruby gem and macOS or Windows installer, is ideal for sysadmins and others who want to perform a wide range of automation tasks on remote bare-metal servers, VMs or cloud instances without the need for any prerequisites. Puppet Bolt doesn't require any previous Puppet know-how. Nor does it require a Puppet agent or Puppet master. It uses only SSH and WinRM (or can piggyback Puppet transports) to communicate and execute tasks on remote nodes.

Despite its simplicity, Puppet Bolt can execute all your existing scripts written in Bash, PowerShell, Python or any other language, stop and start Linux or Windows services, gather information about packages and system facts, or deploy procedural orchestrated workflows, otherwise known as plans. You can do all this right from your workstation or laptop.

For those already using open-source Puppet or Puppet Enterprise, Puppet Bolt enables you to take advantage of the more than 5,700 modules available in the Puppet Forge for everything from deploying database servers to setting up Docker or Kubernetes. You also can query PuppetDB directly with Puppet Bolt.

Install Puppet Bolt and Run Some Tasks

You also can install Puppet Bolt with `apt` or `yum` once you add the Puppet repositories:

```
$ sudo apt install puppet-bolt
```

You can install Puppet Bolt on Windows with the available .msi, or if you're running Bash on Windows 10, by using the Linux instructions for the flavor you installed. Follow the link in the Resources section to see detailed installation instructions for your favorite platform.

If you're running Ruby (and have `gcc` and `make` on your workstation), you can get Puppet Bolt up and running in moments with the simple command:

```
$ gem install bolt
```

In just a few minutes, you're now ready to start running one-off commands, tasks, scripts or plans. Puppet Bolt is perfect for troubleshooting or deploying quick changes, distributing scripts to run across your infrastructure, or automating changes that need to happen in a particular order as part of an application deployment. See the built-in Puppet Bolt commands by running:

```
$ bolt help
```

A typical Puppet Bolt command looks like this:

```
$ bolt <SUBCOMMAND> <ACTION> [options] --nodes <NODE>
```

Where `<SUBCOMMAND>` can be a command, file, script, task or plan. Target nodes can

```
Usage: bolt <subcommand> <action> [options]

Available subcommands:
    bolt command run <command>       Run a command remotely
    bolt file upload <src> <dest>    Upload a local file
    bolt script run <script>         Upload a local script and run it remotely
    bolt task show                   Show list of available tasks
    bolt task show <task>            Show documentation for task
    bolt task run <task> [params]    Run a Puppet task
    bolt plan show                   Show list of available plans
    bolt plan show <plan>            Show details for plan
    bolt plan run <plan> [params]    Run a Puppet task plan
```

Figure 1. Built-in Puppet Bolt Commands

be listed after the `--nodes` (or `-n`) flag or listed in a plain-text file. For example, check the uptime on all your nodes at once like this:

```
$ bolt command run uptime -n server01,server02,server03...
```

Or:

```
$ bolt command run uptime -n @mynodes.txt
```

When executing on WinRM nodes, indicate the WinRM protocol in the nodes string:

```
$ bolt command run <COMMAND> --nodes winrm://<WINDOWS.NODE>
 ↪--user <USERNAME> --password <PASSWORD>
```

Running your existing tried and true scripts (written in any language) on remote nodes is just as simple:

```
$ bolt script run mypythonfile.py -n @mynodes.txt
```

Puppet Bolt really shines when it comes to tasks and plans that rely on more formal orchestration capabilities. You can view built-in Bolt tasks by executing `$ bolt task show` (Figure 2).

```
apply::resource     Apply a single Puppet resource
facts               Gather system facts
facts::bash
facts::powershell
facts::ruby
package             Manage and inspect the state of packages
puppet_conf         Inspect puppet agent configuration settings
service             Manage and inspect the state of services
service::linux      Manage the state of services (without a puppet agent)
service::windows    Manage the state of Windows services (without a puppet agent)
```

Figure 2. Built-in Bolt Tasks

For example, if you want to stop, start or restart services on a Linux or Windows node, you could execute the following task:

```
$ bolt task run service::linux name=cron action=restart -n
 ↪linuxnode01
```

Or:

```
$ bolt task run service::windows name=Netman action=restart
 ↪-n @windowsnodes.txt
```

You also can use the `task` command to look at the status and version of a particular package or app on one or thousands of nodes:

```
$ bolt task run package name=cron action=status -n linuxnode01
Started on linuxnode01...
Finished on linuxnode01:
  {
   "status": "up to date",
   "version": "3.0pl1-128ubuntu2"
  }
Successful on 1 node: linuxnode01
Ran on 1 node in 2.10 seconds
```

```
aggregate::count
aggregate::nodes
canary
facts
facts::info
facts::retrieve
puppetdb_fact

Use `bolt plan show <plan-name>` to view details and parameters for a specific plan.
```

Figure 3. Built-in Bolt Plans

Similarly, you can view built-in plans with the `$ bolt plan show` statement (Figure 3).

You can run these plans without any Puppet agent (or master), but if you want to take advantage of Puppet modules you've written or downloaded from the Forge, such as one to set up nodejs and connect it to your load balancer, you might do this by combining Puppet Bolt with a target node that already has the Puppet agent installed:

```
$ bolt plan run nodejs::myplan load_balancer=lb.myorg.com
```

Not all modules require the Puppet agent. For example, Puppet Bolt can run the **`puppetlabs/aws`** module without it.

Puppet Bolt Is a Great On-Ramp to Puppet

Puppet Bolt is lightweight and easy to learn, with new features added regularly by Puppet's development team, making it an ideal replacement for Ansible, Rundeck, Bladelogic and even Bash and PowerShell scripts. Unlike tools with limited scope, Puppet Bolt treats every target node as a first-class citizen and lets you do everything from quick fixes to full app deployments.

At the same time, working with Puppet Bolt is a great way to introduce yourself to model-based automation, which Puppet pioneered and has become the de facto standard for organizations determined to reap the benefits of pervasive automation. You can use everything you learn with Puppet Bolt elsewhere in the Puppet universe,

which means you and your DevOps teammates finally can share a common tool and expertise—without any prerequisites.

Resources

Hands-on Lab

Puppet Bolt Documentation

Puppet Bolt Information Page

John Tonello is Sr. Technical Marketing Manager for Puppet Inc. He's been a Linux user and enthusiast since building his first Slackware system from diskette 20 years ago.

Road to RHCA: Bumps and Bruises and What I'm Studying

Did you ever crash into another kid when running around on the playground and get a nasty bump—the kind that swells and turns hues of purple, green then yellow? That's how I've felt on many occasions during my journey to Red Hat Certified Architect, although of course, most of my bumps and bruises weren't physical ones, but boy, the emotional and mental ones were not just my imagination.

Some of those bumps and bruises could have completely stopped me, but they've also made me much more resilient. I hope that you too can learn how to build resilience in your life when preparing for exams and certifications or just keeping up with the demands of a career in information technology. My other hope is to inspire others, and especially those who look like me and have chosen a field that can be constantly challenging and exciting but that requires a lot of grit and determination.

Once I decided to pursue the RHCA, I knew it would require time away from family, friends and from doing things that many would consider to be much more fun and exciting than reading, studying and practicing for an exam. I decided first to purchase the exam vouchers and set a date and time that I would take the exam to make me stay on track. I heard of someone completing all five exams in five months, and of course, I think to myself, wow that's amazing. But, I remembered not to compare myself to others, so I returned my focus to what would work best for me.

Realizing I can't predict everything that might happen along the way, I decided to schedule each exam 2–3 months out, depending on what I already know about the subject matter or the practical experience I already have.

In my preparation for the EX407K, the Red Hat Certified Specialist in Ansible Automation, I decided to use Linux Academy's preparation course, "Red Hat Certified Specialist in Ansible Automation Prep Course". The course is 31:19:44 hours long. The great thing about using Linux Academy is that I've used that platform for many years to prepare for other exams, so I know how reliable it is. Also, a great scheduling function is built in to the platform that lets me block out the amount of time and the days I want to study. For this exam, I chose two hours Monday–Friday and three hours on Saturdays. Right now, I'm on track to complete the course by July 14, 2018.

The course includes a lot of quizzes and practice exams. In fact, the practice exam at the end of the course is 2 hours and 22 minutes long. It will allow me to get a sense of where I stand and still give me a couple weeks to take the practice exam several times before the scheduled exam on July 31, 2018.

Listed below are the core skills and abilities I'll possess once I've completed my Linux Academy training, along with some additional reading using the DO407 "Automation with Ansible I" course study guide. I plan to check off these areas as I go along and revisit them at the end of my practice exams:

- Understand the core components of Ansible: inventories, modules, variables, facts, plays, playbooks and configuration files.
- Run ad hoc Ansible commands.
- Use both static and dynamic inventories to define groups of hosts.
- Utilize an existing dynamic inventory script.
- Create Ansible plays and playbooks, which includes 1) knowing how to work with commonly used Ansible modules, 2) using variables to retrieve the results of running a command, 3) using conditionals to control play execution, 4) configuring error handling, 5) creating playbooks to configure systems to a specified state, and 6) selectively running specific tasks in playbooks using tags.

- Create and use templates to create customized configuration files.
- Work with Ansible variables and facts.
- Create and work with roles.
- Download roles from an Ansible Galaxy and use them.
- Manage parallelism.
- Use Ansible Vault in playbooks to protect sensitive data.
- Install Ansible Tower and use it to manage systems.
- Use the provided documentation to look up specific information about Ansible modules and commands.

Because the EX407 is performance-based like all of Red Hat's exams, practice, practice, practice is necessary. The exam likely will require me to develop Ansible playbooks that configure systems for specific roles and then apply those playbooks to systems to implement those roles. Also, I may be asked to demonstrate my ability to run Ansible playbooks and configure an Ansible environment for specific behaviors.

The exam is three hours. Official scores for the exam will come from Red Hat, and of course, I'll sign a non-disclosure agreement during the exam.

Earning the Red Hat Certificate of Expertise in Ansible Automation may be good for the following:

- System administrators who need to manage large numbers of systems.
- System administrators who work in a DevOps environment and want to automate a large part of their day-to-day workload.

- Developers who have a basic systems administration background and want to incorporate automation into their development process.

- Red Hat Certified Engineers (RHCE) interested in earning a Red Hat Certificate of Expertise or RHCA credential.

So although on some days, I may not complete my study schedule because of other things life brings, I know adjustments are always possible. The point is not to get down on myself but to focus on the end goal. And, now I'm off to study Loops and Conditionals, error handling in playbooks, tagging tasks in playbooks and then do a quiz, a couple exercises and a couple hands-on labs. Wish me luck.

—Taz Brown

News Briefs

Visit LinuxJournal.com for daily news briefs.

- The 2018 Open Source Job Report is now available from The Linux Foundation and Dice. Some key findings include: "Linux is back on top as the most in-demand open source skill category, making it required knowledge for most entry-level open source careers" and "Containers are rapidly growing in popularity and importance, with 57% of hiring managers seeking that expertise, up from only 27% last year."

- Ubuntu started collecting user data with version 18.04 (users can opt out during the install), and the first report is now available. According to the report, 67% of users opt in, installation takes 18 minutes, most people are installing from scratch instead of upgrading and having a single CPU is most common. In addition, the report reveals that although the US has the highest concentration of users, Brazil, India, China and Russia also are big Ubuntu users.

- Oracle has started charging for Java SE and support. According to *The Register*, the cost for the "Java subscription" is "$25 per processor per month, and $2.50 per user per month on the desktop, or less if you buy lots for a long time." The article notes that "If you like your current Java licences, Oracle will let you keep them." But also that "come January 2019 Java SE 8 on the desktop won't be updated any more...unless you buy a sub."

- The EFF announced the launch of STARTTLS Everywhere, "EFF's initiative to improve the security of the email ecosystem". The goal with STARTTLS is "to do for email what we've done for web browsing: make it simple and easy for everyone to help ensure their communications aren't vulnerable to mass surveillance." You can find out how secure your current email provider is at https://www.starttls-everywhere.org, and for a more technical deep dive into STARTTLS Everywhere, go here.

- GitLab announced that it is moving from Azure to Google Cloud. GitLab claims the decision to switch to Google Cloud is "because of our desire to run GitLab on Kubernetes. Google invented Kubernetes, and GKE has the most robust and mature

Kubernetes support." The migration is planned for Saturday, July 28, 2018, and GitLab will utilize its Geo product for the migration.

- Mercedes-Benz Vans has adopted the Automotive Grade Linux open platform, automotiveIT reports. The company plans to use the OS in its upcoming commercial vehicles, the first of which will be seen in prototype later this year. Thomas Wurdig, head of onboard system architecture and IoT, Mercedes-Benz Vans, stated "Using a standardized, open operating system like AGL enables us to rapidly develop new commercial vehicle use cases such as robotic delivery, data analytics, and prediction and automation technologies."

- Python 3.7.0 has been released. This is a major release of the Python language, containing many new features, including new syntax features, backwards-incompatible syntax changes, new library modules, significant improvements to the standard library and much more. See the release highlights for all the updates, and go here to download.

- Red Hat OpenStack Platform 13 is now available. This is a "long-life release", which comes with up to three years of standard support, and optional two years of extended life-cycle support. This release has many new features, including "fast forward upgrades", and it now supports containerization of all OpenStack services. OpenStack Platform 13 also delivers several new integrations with Red Hat OpenShift Container Platform, and it has many new hardened security services. See the Red Hat blog post for more information.

- Eighteen Chromebooks from Acer, Asus, Lenovo and Dell—all based on Intel Apollo Lake—to receive Linux app support. According to xda Developers, "as the change has only just landed, Canary and Developer channels will see this first in the coming days and weeks. Stable or Beta channel users will have to wait until Chrome OS version 69."

- SUSE is being acquired by EQT. SUSE.com notes that with this partnership "SUSE expects to be equipped to further exploit the excellent market opportunity both in the Linux operating system area as well as in emerging product groups in the open

source space." SUSE CEO Nils Brauckmann will continue to lead SUSE, and "the SUSE business expects staffing, customer relationships, partnerships, product and service offering, commitment to open source leadership and support for the key open source communities to remain unchanged."

- The Linux Foundation recently announced that Google has become a Platinum Member of the foundation. From the press release: "'Google is one of the biggest contributors to and supporters of open source in the world, and we are thrilled that they have decided to increase their involvement in The Linux Foundation,' said Jim Zemlin, executive director, The Linux Foundation. 'We are honored that Sarah Novotny, one of the leading figures in the open source community, will join our board—she will be a tremendous asset.'"

- The Dell Precision 7530 and 7730 Mobile Workstation Developer Editions are now available via Dell's online store with Ubuntu Linux preinstalled, Softpedia News reports. The Mobile Workstations are powered by the latest Intel Core or Xeon processors, and "feature blazing-fast RAM, professional AMD or Nvidia graphics cards, and are certified for the Red Hat Enterprise Linux 7.5 operating system". Prices for the "world's most powerful 15" and 17" laptops with Ubuntu pre-installed" begin at $1,091.14 for the 7530 and $1,371.37 for the 7730.

- Canonical released its new Minimal Ubuntu. According to the Ubuntu blog, Minimal Ubuntu is "optimized for automated use at scale, with a tiny package set and minimal security cross-section. Speed, performance and stability are primary concerns for cloud developers and ops." The images are 50% smaller than the standard Ubuntu server images and they boot up to 40% faster. Minimal Ubuntu also is fully compatible with standard Ubuntu operations. You can download it here.

- The Xen Hypervisor 4.11 was released. In this release "PVH Dom0 support is now available as experimental feature and support for running unmodified PV guests in a PVH Container has been added. In addition, significant chunks of the ARM port have been rewritten." Xen 4.11 also contains mitigations for Meltdown and Spectre vulnerabilities. For detailed download and build instructions, go here.

- There's a new text-based browser called Browsh, Phoronix reports. Browsh can render anything a modern browser can, and you can use it from a terminal or within a normal browser to reduce bandwidth and increase browser speed. For more info and to download, see the Browsh project website.

- Sirin Labs to launch the $1,000 Finney cryptocurrency smartphone this fall, Engadget reports. The Finney (named after Bitcoin pioneer Hal Finney) is a "state of the art mobile device for the blockchain era" and runs on a forked version of Android. It has a slider on the back where "you'll find a secondary display, called the Safe Screen, that's only used for crypto transactions....The slider also activates the cold storage wallet that is designed to hold a significant number of different cryptocurrencies."

- freenode has a new job board. jobs.freenode.net "aims to connect those looking to hire with the immense talent that can be found within the wider freenode communities". The job board is free to use, but companies that use it successfully are encouraged to make a donation to help support the freenode network, jobs.freenode.net and the annual freenode #live conference.

- Python's Benevolent Dictator For Life (BDFL) Guido van Rossum announced he's stepping down from the role. On the Python mailing list, van Rossum said, "I would like to remove myself entirely from the decision process. I'll still be there for a while as an ordinary core dev, and I'll still be available to mentor people—possibly more available. But I'm basically giving myself a permanent vacation from being BDFL, and you all will be on your own." He credits his decision to step down as partly due to his experience with the turmoil over PEP 572: "Now that PEP 572 is done, I don't ever want to have to fight so hard for a PEP and find that so many people despise my decisions." van Rossum says he will not appoint a successor and leaves that to the development team to decide upon. For old-time's sake, see *Linux Journal*'s interview with Guido van Rossum from 1998.

Cleaning Your Inbox with Mutt

Kyle Rankin is a Tech Editor and columnist at *Linux Journal* and the Chief Security Officer at Purism. He is the author of *Linux Hardening in Hostile Networks*, *DevOps Troubleshooting*, *The Official Ubuntu Server Book*, *Knoppix Hacks*, *Knoppix Pocket Reference*, *Linux Multimedia Hacks* and *Ubuntu Hacks*, and also a contributor to a number of other O'Reilly books. Rankin speaks frequently on security and open-source software including at BsidesLV, O'Reilly Security Conference, OSCON, SCALE, CactusCon, Linux World Expo and Penguicon. You can follow him at @kylerankin.

Teach Mutt yet another trick: how to filter messages in your Inbox with a simple macro.

By Kyle Rankin

I'm a longtime Mutt user and have written about it a number of times in *Linux Journal*. Although many people may think it's strange to be using a command-line-based email client in 2018, I find a keyboard-driven email client so much more efficient than clicking around in a web browser. Mutt is extremely customizable, which presents a steep learning curve at first, but now that I'm a few decades in, my Mutt configuration is pretty ideal and fits me like a tailored suit.

Of course, as with any powerful and configurable tool, every now and then I learn of a new Mutt feature that improves my quality of life dramatically. In this case, I was using an email system that didn't offer server-side filters. Because I was a member of many different email groups and aliases, this meant that my Inbox was flooded with emails of all kinds, and it became difficult to filter through all the unimportant email I wanted to archive with the emails that demanded my immediate attention.

There are many ways to solve this problem, some of which involve tools like offlineimap combined with filtering tools. With email clients like Thunderbird, you also can set up filters that automatically move email to other folders every time you sync. I wanted a similar system with Mutt, except I didn't want it to happen automatically. I wanted to be able to press a key first so

I could confirm what was moving. In the process of figuring this out, I discovered a few gotchas I think other Mutt users will want to know about if they set up a similar system.

Tagging Emails

The traditional first step when setting up a keyboard macro to move email messages based on a pattern would be to use Mutt's tagging-by-pattern feature (by default, the T key) to tag all the messages in a folder that match a certain pattern. For instance, if all of your cron emails have "Cron Daemon" in the subject line, you would type the following key sequence to tag all of those messages:

```
TCron Daemon<enter>
```

That's the uppercase T, followed by the pattern I want to match in the subject line (Cron Daemon) and then the Enter key. If I type that while I'm in my Mutt index window that shows me all the emails in my Inbox, it will tag all of the messages that match that pattern, but it won't do anything with them yet. To act on all of those messages, I press the ; key (by default), followed by the action I want to perform. So to save all of the tagged email to my "cron" folder, I would type:

```
;s=cron<enter>
```

That's ; followed by the s key to save, followed by the name of the folder to save to, where `=cron` means "the folder named cron that sits under the Inbox". To combine all of this into a macro so I can trigger this action by pressing, say, .c, I would add the following to my Mutt configuration file:

```
macro index .c "TCron Daemon<enter>;s=cron<enter>"
```

Or, if you want to make it more portable (in case you remapped your save command to another key), you could do this:

```
macro index .c "TCron Daemon<enter><tag-prefix>
↪<save-message>=cron<enter>"
```

Of course, if you're cleaning out a lot of messages in your Inbox, you'll probably have a lot of different patterns to match. For instance, I want to move all of the DMARC report messages that are sent to the dmarc-reports email address, so I'm going to add another pattern to this macro that will save all of those messages to my dmarc folder. By itself, the macro would look like this:

```
macro index .c "T~Cdmarc-reports<enter><tag-prefix>
↪<save-message>=dmarc<enter>"
```

The most important difference here is that for my tagging pattern, instead of just matching on `dmarc-reports`, which would match only the subject line, I typed `~C` in front of it, which tags all messages that have "dmarc-reports" in the To: or CC: headers. The combined macro just combines the two lists of keypresses one after the other and looks like this:

```
macro index .c "TCron Daemon<enter><tag-prefix><save-message>
↪=cron<enter>T~Cdmarc-reports<enter><tag-prefix><save-message>
↪=dmarc<enter>"
```

The Problem

The above macro has a subtle problem though, and unless you have set up a Mutt macro like this in the past, you may not notice it. In fact, the first couple times you run the macro, it may seem like it works—as long as there are matching messages in your Inbox. The problem occurs if you don't have any matching messages. The way that Mutt interprets this macro, if you don't have any matching messages, it still will happily apply any commands that follow the <tag-prefix> command—only to whatever message the cursor is currently under! Luckily I was just moving messages around, but if you told Mutt to delete tagged messages, they would be gone for good!

The solution here is to use a special Mutt command called `<tag-prefix-cond>` instead of `<tag-prefix>`. This tells Mutt to execute the command following `<tag-prefix-cond>` only if Mutt actually has any messages tagged. You then wrap the command with `<end-cond>` to tell Mutt that conditional command is completed.

So for a simple macro, I would replace:

```
macro index .c "TCron Daemon<enter><tag-prefix><save-message>
↪=cron<enter>"
```

with:

```
macro index .c "TCron Daemon<enter><tag-prefix-cond>
↪<save-message>=cron<enter><end-cond>"
```

As you can see, I wrapped the entire `<save-message>` command up inside this conditional block. If I gave the same treatment to the full macro, I would convert:

```
macro index .c "TCron Daemon<enter><tag-prefix><save-message>
↪=cron<enter>T~Cdmarc-reports<enter><tag-prefix>
↪<save-message>=dmarc<enter>"
```

to:

```
macro index .c "TCron Daemon<enter><tag-prefix-cond>
↪<save-message>=cron<enter><end-cond>T~Cdmarc-reports<enter>
↪<tag-prefix-cond><save-message>=dmarc<enter><end-cond>"
```

Now when I want to add a new filter for my Inbox, I can test the `tag` command one time within Mutt by hand to confirm it does what I expect, and then append it to my macro. And now when I load my Inbox, I can press a simple key and perform those filters. If you want this to be automatic, you would just set up a folder-hook statement for your Inbox folder that uses the Mutt `push` command to press all of the keys in the above macro. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

Python and Its Community Enter a New Phase

Reuven M. Lerner teaches Python, data science and Git to companies around the world. His free, weekly "better developers" email list reaches thousands of developers each week; subscribe here. Reuven lives with his wife and children in Modi'in, Israel.

On Python's BDFL Guido van Rossum, his dedication to the Python community, PEP 572 and hope for a healthy outcome for the language, open source and the computing world in general.

By Reuven M. Lerner

Python is an amazing programming language, there's no doubt about it. From humble beginnings in 1991, it's now just about everywhere. Whether you're doing web development, system administration, test automation, devops or data science, odds are good that Python is playing a role in your work.

Even if you're not using Python directly, odds are good that it is being used behind the scenes. Using OpenStack? Python plays an integral role in its development and configuration. Using Dropbox on your computer? Then you've got a copy of Python running on your computer. Using Linux? When I purchased Red Hat Linux back in 1995, the configuration was a breeze—thanks to visual tools developed in Python.

And, of course, there are numerous schools and educational programs that are now teaching Python. MIT's intro computer science course switched several years ago from Scheme to Python, and thousands of universities all over the world made a similar switch in its wake. My 15-year-old daughter participates in a program for technology and entrepreneurship—and she's learning Python.

There currently is an almost insatiable demand for Python developers. Indeed, Stack Overflow reported last year that Python is not only the most popular language on its site, but it's also the fastest-growing language. I can attest to this popularity in my own job as a freelance Python trainer. Some of the largest computer companies in the world are now using Python on a regular basis, and their use of the language is growing, not shrinking.

Normally, a technology with this much impact would require a large and active marketing department. But Python is (of course) open-source software, and its success is the result of a large number of contributors—to the core language, to its documentation, to libraries and to the numerous blogs, tutorials, articles and videos available online. I often remind my students that people often think of "open source" as a synonym for "free of charge", but that they should instead think of it as a synonym for "powered by the community"—and there's no doubt that the Python community is strong.

Such a strong community doesn't come from nowhere. And there's no doubt that Guido van Rossum, who created Python and has led its development ever since, has been a supremely effective community organizer and leader.

Programmers often think about code more than people and efficiency more than empathy. We tend to be hot-headed, jumping headlong into debates over obscure technical points. Guido (as he is universally known among Python developers) struck a different tone in his management of Python. From the beginning, he thought about who would use the language and what mistakes they were apt to make—and how the language could best serve such people. He did debate other developers over features, bugs and the language's direction, but he generally did so through respect

and reasoned argument, rather than with dismissive, expletive-filled tirades that are so common in the Open Source world.

In this way, Guido managed to build not only a great language, but also a culture of giving, sharing and helping. We shouldn't take this for granted. Although programming can be hard, managing other people and taking their varying perspectives into account is even harder.

Over time, this culture led to many people participating in the Python language and ecosystem. When I was at PyCon in Cleveland this year, it was hard not to be amazed by the large number of people, from all over the world, using Python for a wide variety of tasks. Guido himself didn't even speak at the conference. I hope that he was satisfied to walk around, taking in the awesome sight of thousands of people passionate about the language that he had created.

Perhaps even more impressive to me was the fact that hundreds of people remained for the open-source sprint days following PyCon. Almost all of those people were volunteering their time to improve some part of the Python language they used every day. This dedication, coupled with an interest in contributing and helping others, is a hallmark of open source in general, and of Python in particular.

Today, Python has not only a strong community, but a robust organizational structure as well. The Python Software Foundation manages funds that can help developers to contribute to the language. The core developers are impressively organized, coordinating releases large and small, and providing stability and clarity that easily rivals commercial software houses.

But while Python's bureaucracy and ecosystem have grown over time, one man has still stood at the center of it all: Guido van Rossum, who was dubbed the BDFL, or Benevolent Dictator For Life. Although this means that Guido has (deservedly) received a great deal of praise, it means he was the target of numerous slings and arrows as well.

Perhaps the biggest dispute in the Python world was the transition from version 2 to version 3. Actually, I shouldn't even say "was" here, since many of my clients are still making that change. It's easy to say that this incompatible upgrade was handled poorly. And you can be sure that the Python community has learned its lesson and will never make such a break again. However, the end result—Python 3—is definitely better than Python 2. Should Guido and the core developers have handled it differently? Perhaps, but it's easy to say that in retrospect. The number of blog posts calling Python 3 a colossal mistake must have hurt Guido. But part of being a manager is having the ability to accept, filter and even ignore criticism, and he did it with aplomb.

Decisions regarding changes to Python have long been made via PEPs (Python Enhancement Proposals)—documents in which new ideas are proposed, kicked around, modified and finally approved or rejected. As the BDFL, Guido's input and approval always were needed for a PEP to be considered "approved".

Through the years, I've learned to trust Guido's judgment. There were numerous cases when I would read a PEP, roll my eyes, and think, "Really? Isn't there a better way to do this?" But over time, I would generally say, "Hmm, maybe he was right." Guido was not only a talented developer and manager, but also someone with taste and vision, whose experience allowed him to sense better than many others in the community what was the right direction for the language.

And, then we got PEP 572, for "assignment expressions". Python distinguishes between statements (which perform an action) and expressions (which return values). An "if" statement must contain an expression, and an assignment is a statement, which means that you cannot say:

```
x = 5
while x = x - 1:
    print(f"x is {x}")
```

The above, as a statement, would be flagged as an error by Python before it was

even executed.

PEP 572 introduced a new syntax, the "assignment expression", which would allow the above to work. That PEP recently was approved by Guido and will be a part of Python 3.8. It'll look like this:

```
x = 5
while x := x - 1:
    print(f"x is {x}")
```

It turns out that a huge number of Python developers, including some of the core developers, disliked this. In fact, they disliked this *a lot*. I must say, I'm still not enamored of it, but I'm applying my usual "defer-to-Guido" thinking here, believing that within a few years, I'll agree that assignment expressions are a good and useful thing to have in Python. After all, I've already changed my tune on str.format, enums and type annotations; perhaps over time, I'll agree with this one as well.

A large number of Python developers decided not to hold off on their judgment and attacked PEP 572 as completely out of line, or even "un-Pythonic", a strong phrase to be leveled at the guy who has dictated Python's development and direction for nearly three decades.

These accusations have taken their toll. July 12, 2018, Guido sent a message to the python-committers e-mail list, saying that he's taking a "permanent vacation from being BDFL". He further says, "You will all be on your own."

On the one hand, it's terribly sad that the discussion over PEP 572 become so heated, so uncharacteristically un-Pythonic in its nature, that Guido (of all people!) found himself overwhelmed and unhappy. His vision and input will be missed, although he does point out that he'll continue to be a core contributor. And you can be sure that he'll have a lot to say, and that we would be wise to listen to it.

But on the other hand, perhaps we should see this as something of an

opportunity for the Python community to mature. Guido has been running a several-thousand-person, worldwide development effort for several decades, in which most of his developers are volunteers. That's no small feat, and he deserves a chance to reset, or even to retire. I mean, how long can you expect someone to do this kind of work?

Moreover, the Python community can and should evolve beyond the BDFL style of leadership. Guido himself is curious to see what's going to happen, as he writes: "So what are you all going to do? Create a democracy? Anarchy? A dictatorship? A federation?"

The fact is that while benevolent dictatorships are (by definition) good, they aren't sustainable. And thus, the Python community will now not have to grow, so much as evolve—becoming an institution that can outlive any one of its individual contributors, while continuing to have a massive impact on the world.

Guido is, in my mind, an inspiring developer, leader and manager. What he created has affected countless people around the world. I hope and expect that the future of the Python community will reflect the strength and wisdom of its founder, and that it'll continue to have the same impact as it enters this new phase. This transition period will likely be somewhat messy, but it'll also be healthy—and the end result will be good for Python, good for open source, good for developers, and good for everyone who uses computers. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

Creating the Concentration Game PAIRS with Bash

Exploring the nuances of writing a pair-matching memory game and one-dimensional arrays in Bash.

Dave Taylor has been hacking shell scripts on Unix and Linux systems for a really long time. He's the author of *Learning Unix for Mac OS X* and *Wicked Cool Shell Scripts*. You can find him on Twitter as @DaveTaylor, and you can reach him through his tech Q&A site Ask Dave Taylor.

I've always been a fan of Rudyard Kipling. He wrote some great novels and stories, mostly about British colonial-era India. Politically correct in our modern times? Not so much, but still, his books are good fun for readers and still are considered great literature of its time. His works include *The Jungle Book*, *Captains Courageous*, *The Just So Stories* and *The Man Who Would Be King*, among many others.

He also wrote a great spy novel about a young English boy who is raised as an Indian native and thence recruited by the British government as a spy. The boy's name is the title of the book: *Kim*. In the story, Kim is trained to have an eidetic memory with a memory game that involves being shown a tray of stones of various shapes, sizes and colors. Then it's hidden, and he has to recite as many patterns as he can recall.

For some reason, that scene has always stuck with me, and I've even tried to teach my children to be situationally aware

through similar games like "Close your eyes. Now, what color was the car that just passed us?" Since most of us are terrible observers (see, for example, how conflicting eyewitness accident reports can be), it's undoubtedly good practice for general observations about life.

Although it's tempting to try to duplicate this memory game as a program, the reality is that with just a shell script, it would be difficult. Perhaps you display a random pattern of letters and digits in a grid, then clear the screen, then ask the user to enter patterns, but that's really much more of a game for a screen-oriented, graphical application—not shell scripts.

But, there's a simplified version of this that you can play with a deck of cards: Concentration. You've probably played it yourself at some point in your life. You place the cards face down in a grid and then flip up two at a time to try to find pairs. At the beginning, it's just random guessing, but as the game proceeds, it becomes more about your spatial memory, and by the end, good players know what just about every unflipped card is at the beginning of their turn.

Designing PAIRS

That, of course, you can duplicate as a shell script, and since it is going to be a shell script, you also can make the number of pairs variable. Let's call this game PAIRS.

As a minimum, let's go with four pairs, which should make debugging easy. Since there's no real benefit to duplicating playing card values, it's just as easy to use letters, which means a max of 26 pairs, or 52 slots. Not every value is going to produce a proper spread or grid, but if you aim for 13 per line, players then can play with anywhere from 1–4 lines of possibilities.

Playability would be enhanced by having clickable spots, but this is a shell script, dude, so you'll have to suffer through a Chess-style grid notation: line,slot.

A display could look something like this mockup:

```
    1   2   3   4   5   6   7   8   9   10  11  12  13
1: [-] [-] [-] [-] [-] [-] [-] [-] [-] [-] [-] [-] [-]
2: [-] [-] [-] [A] [-] [-] [-] [-] [-] [-] [-] [-] [-]
3: [-] [-] [-] [-] [-] [-] [-] [-] [E] [-] [-] [-] [-]
4: [-] [-] [-] [-] [-] [-] [-] [-] [-] [-] [-] [-] [Z]
```

The letter A is at 2,4, and the letter E is at 3,9. Even better though, because the first value should only ever be 1–4, you could omit the comma to make things more succinct, making the A at 24 and the E at 39. There'll be three-digit values too, like the Z at 413, but that's still easy to pull apart and process.

Actually, now that there's a 13x4 grid pattern, let's rethink the flexibility of the game. Instead of allowing an arbitrary number of pairs, let's make the game less flexible and constrain users to being able to specify only how many rows of 13 they want. Even more so, only even numbers of rows will be acceptable. (Obviously with a single row of 13, it'd be hard to have only pairs show up!)

Data Representation

Bash has decent support for arrays, so my first inclination is to make this a linear array and simply divide by 13, as needed to convert "display" coordinates to actual array coordinates. A multi-dimensional array would be better, but Bash doesn't offer any support for that particular data structure. There's a workaround by using an associative array, but that requires Bash 4.x, and MacOS X still ships with Bash 3.x, which means a lot of our readers (shhh, I know not all of you are full-time Linux folk) would be left out in the cold. Ah, fun, fun.

Enough chat though, let's get to the coding side of things, although I'll have to continue the development in my next column.

Coding the Grid

A reasonable place to start this whole project is by looking at how to work with a one-dimensional array in Bash. An array slot is assigned a value like this:

```
myArray[2]=value
```

And it's referenced in the slightly clumsy format:

```
echo ${myArray[2]}
```

Bash wants you to use the declare statement to identify arrays prior to use—`declare -a myArray`—and you can assign initial values like this:

```
declare -a myArray=(cat dog pig frog snake)
```

There are some handy shortcuts with @ and * to learn about, but let's just start coding.

First things first, here's the Bash function I've written to initialize the array with A–Z values (I'll shuffle the values in a different function):

```
initialize ()
{
   # initialize the board with sequential letters

   count=1   maxcount=$1

   while [ $count -le $maxcount ]
   do
     addon=$(( 13 * ( $count - 1 ) ))

     for slot in {1..13}
     do
       index=$(( $addon + $slot ))
       letter=$(( $index % 26 ))
       board[ $index ]=${letters[$letter]}
     done
     count=$(( $count + 1 ))
   done
}
```

While it may appear that there's lots to consider here, the heart of the function is the `for` loop within a `while` loop. The `while` steps through rows, and the `for` loop steps through the 13 slots for each row.

The most interesting line might be this:

```
board[ $index ]=${letters[$letter]}
```

The value of a given slot in the board array is going to be set to the `$letter` index value of the `$letters` array. That array is set in the opening declarations:

```
declare -a letters=(A B C D E F G H I J K L M N O P Q R
                    S T U V W X Y Z)
```

The result is interesting, because Bash really wants to have 0-based indexing, and here I'm using it all as 1-based indexing instead. But, that I shall show you in my next article, the continuation of the PAIRS game-programming adventure! ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

The freenode project celebrates its 20th anniversary this year at the second annual freenode #live conference

At We The Curious in Bristol, UK
November 3-4, 2018 — 9am Saturday - 6pm Sunday

Keynote speakers include Bradley M. Kuhn, Chris Lamb, Kyle Rankin, Leslie Hawthorn and VM Brasseur! More to come...

Registration and call for participation is open now at

HTTPS://FREENODE.LIVE

What's New in Kernel Development

By Zack Brown

Zack Brown is a tech journalist at *Linux Journal* and *Linux Magazine*, and is a former author of the "Kernel Traffic" weekly newsletter and the "Learn Plover" stenographic typing tutorials. He first installed Slackware Linux in 1993 on his 386 with 8 megs of RAM and had his mind permanently blown by the Open Source community. He is the inventor of the *Crumble* pure strategy board game, which you can make yourself with a few pieces of cardboard. He also enjoys writing fiction, attempting animation, reforming Labanotation, designing and sewing his own clothes, learning French and spending time with friends'n'family.

New Intel Caching Feature Considered for Mainline

These days, **Intel**'s name is Mud in various circles because of the **Spectre/Meltdown** CPU flaws and other similar hardware issues that seem to be emerging as well. But, there was a recent discussion between some Intel folks and the kernel folks that was not related to those things. Some thrust-and-parry still was going on between kernel person and company person, but it seemed more to do with trying to get past marketing speak, than at wrestling over what Intel is doing to fix its longstanding hardware flaws.

Reinette Chatre of Intel posted a patch for a new chip feature called **Cache Allocation Technology** (CAT), which "enables a user to specify the amount of cache space into which an application can fill". Among other things, Reinette offered the disclaimer, "The cache pseudo-locking approach relies on generation-specific behavior of processors. It may provide benefits on certain processor generations, but is not guaranteed to be supported in the future."

Thomas Gleixner thought Intel's work looked very interesting

and in general very useful, but he asked, "are you saying that the CAT mechanism might change radically in the future [that is, in future CPU chip designs] so that access to cached data in an allocated area which does not belong to the current executing context won't work anymore?"

Reinette replied, "Cache Pseudo-Locking is a model-specific feature so there may be some variation in if, or to what extent, current and future devices can support Cache Pseudo-Locking. CAT remains architectural."

Thomas replied, "that does NOT answer my question at all."

At this point, **Gavin Hindman** of Intel joined the discussion, saying:

> Support in a current generation of a product line doesn't imply support in a future generation. Certainly we'll make every effort to carry support forward, and would adjust to any changes in CAT support, but we can't account for unforeseen future architectural changes that might block pseudo-locking use-cases on top of CAT.

And Thomas replied, "that's the real problem. We add something that gives us some form of isolation, but we don't know whether next generation CPUs will work. From a maintainability and usefulness POV, that's not a really great prospect."

Elsewhere in a parallel part of the discussion, Thomas asked, "Are there real world use cases that actually can benefit from this [CAT feature] and what are those applications supposed to do once the feature breaks with future generations of processors?"

Reinette replied, "This feature is model-specific with a few platforms supporting it at this time. Only platforms known to support Cache Pseudo-Locking will expose its resctrl interface."

To which Thomas said, "you deliberately avoided to answer my question again."

Gavin replied now, saying:

> Reinette's not trying to avoid the questions, we just don't necessarily have definitive answers at this time. Currently pseudo-locking requires manual setup on the part of the integrator, so there will not be any invisible breakage when trying to port software expecting pseudo-locking to new devices, and we'll certainly do everything we can to minimize user-space/configuration impact on migration if things change going forward, but these are unknowns. We are in a bit of chicken/egg where people aren't broadly using it because it's not architectural, and it's not architectural because people aren't broadly using it. We could publicly carry the patches out of mainline, but our intent for pushing the patches to mainline are to a) increase exposure/usage, b) reduce divergence across people already using hacked versions, and c) ease the overhead in keeping patches in sync with the larger CAT infrastructure as it evolves - we are clear on the potential support burden being incurred by submitting a non-architectural feature, and there's certainly no intent to dump a science-experiment into mainline.

Thomas replied, "Ok. So what you are saying is that 'official' support should broaden the user base, which in turn might push it into the architectural realm. I'll go through the patch set with this in mind."

Elsewhere, Thomas and Reinette went through a more technical exchange of data, and Reinette provided useful data points for understanding the value of the CAT feature itself. To all of this, Thomas said, "Very nice. Thank you so much for doing this. That kind of data is really valuable. My take away from this: All of the mechanisms are only delivering best effort and the real benefit is the reduction of average latency. The worst case outliers are in the same ballpark at seems." And, he promised to review the next version of Intel's patch, which Reinette expected to send out within the week.

So as Intel tries to move past Spectre/Meltdown, it continues to collaborate with kernel developers on this sort of feature. At the same time, it's hard to forget that its hardware problems are not over, and that new CPU flaws continue to be discovered

even now. **Linus Torvalds** has interpreted some of Intel's statements to mean that Intel does not intend to fix some of its hardware flaws in future generations of CPUs, which would force kernel developers, and developers of other operating systems, to work around those flaws for the foreseeable future. So there's a lot of tension even in the context of collaborating on relatively simple new features like CAT.

Supporting the NDS32 Architecture

Green Hu posted a patch to support the **NDS32** architecture. He described the current status as, "It is able to boot to shell and passes most LTP-2017 testsuites in nds32 AE3XX platform."

Arnd Bergmann approved the patch, but Linus Torvalds wanted a little more of a description—an overview of the "uses, quirks, reasons for existing" for this chip, to include in the changelog.

Arnd replied:

> The non-marketing description is that this is a fairly conventional (in a good way) low-end RISC architecture that is usually integrated into custom microcontroller and SoC designs, competing with the similar ARM32, ARC, MIPS32, RISC-V, Xtensa and (currently under review) C-Sky architectures that occupy the same space. The most interesting bit from my perspective is that Andestech is already selling a new generation of CPU cores that are based on 32-bit and 64-bit RISC-V, but are still supporting enough customers on the existing cores to invest in both.

And Green also said:

> Andes' nds32 architecture supports Linux for Andes' N10, D10, N13, N15, D15 processor cores.
>
> Based on the patented 16/32-bit AndeStar RISC-like architecture, we designed the configurable AndesCore series of embedded processor families. AndesCores range from highly performance-efficient small-footprint cores for microcontrollers

> and deeply-embedded applications to 1GHz+ cores running Linux, covering general-purpose N-series cores for a wide range of computing needs; DSP-capable D-series cores for digital signal control; instruction-extensible E-series cores for application-specific acceleration; and secure S-series cores for best protection of the most valuable.
>
> Our customers together have shipped over 2.5 billion SoCs with Andes processors embedded (including non-MMU IP cores). It will help our customers to get better Linux support if we are merged into mainline.

It looks like there's no controversy over this port, and it should fly into the main tree. One reason for the easy adoption is that it doesn't touch any other part of the kernel—if the patch breaks anything, it'll break only that one architecture, so there's very little risk in letting Green make his own choices about what to include and what to leave out. Linus' main threshold will probably be "does it compile?" If yes, then it's okay to go in.

The situation may start to become interesting if other parts of the kernel begin offering special behaviors for the NDS32 architecture, and if those behaviors start deviating too far from other architectures. For example, some architectures have special memory managing features that the kernel proper can take advantage of. Once NDS32 starts influencing code in other parts of the kernel, that likely would be the time Green's patches start to get a lot more scrutiny.

Note: if you're mentioned in this article and want to send a response, please send a message with your response text to ljeditor@linuxjournal.com, and we'll run it in the next Letters section and post it on the website as an addendum to the original article. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

DEEP DIVE

CONTAINERS

Everything You Need to Know about Linux Containers, Part I: Linux Control Groups and Process Isolation

Everyone's heard the term, but what exactly are containers?

By Petros Koutoupis

The software enabling this technology comes in many forms, with Docker as the most popular. The recent rise in popularity of container technology within the data center is a direct result of its portability and ability to isolate working environments, thus limiting its impact and overall footprint to the underlying computing system. To understand the technology completely, you first need to understand the many pieces that make it all possible.

Containers vs. Virtual Machines

People often ask about the difference between containers and virtual machines. Both have a specific purpose and place with very little overlap, and one doesn't obsolete the other. A container is meant to be a lightweight environment that you spin up to host one to a few isolated applications at bare-metal performance. You should opt for virtual machines when you want to host an entire operating system or ecosystem or maybe to run applications incompatible with the underlying environment.

Linux Control Groups

Truth be told, certain software applications in the wild may need to be controlled or limited—at least for the sake of stability and, to some degree, security. Far too often, a bug or just bad code can disrupt an entire machine and potentially cripple an entire ecosystem. Fortunately, a way exists to keep those same applications in check. Control groups (cgroups) is a kernel feature that limits, accounts for and isolates the CPU, memory, disk I/O and network's usage of one or more processes.

Originally developed by Google, the cgroups technology eventually would find its way to the Linux kernel mainline in version 2.6.24 (January 2008). A redesign of this technology—that is, the addition of kernfs (to split some of the sysfs logic)—would be merged into both the 3.15 and 3.16 kernels.

The primary design goal for cgroups was to provide a unified interface to manage processes or whole operating-system-level virtualization, including Linux Containers, or LXC (a topic I plan to revisit in more detail in a follow-up article). The cgroups framework provides the following:

- **Resource limiting:** a group can be configured not to exceed a specified memory limit or use more than the desired amount of processors or be limited

to specific peripheral devices.

- **Prioritization:** one or more groups may be configured to utilize fewer or more CPUs or disk I/O throughput.

- **Accounting:** a group's resource usage is monitored and measured.

- **Control:** groups of processes can be frozen or stopped and restarted.

A cgroup can consist of one or more processes that are all bound to the same set of limits. These groups also can be hierarchical, which means that a subgroup inherits the limits administered to its parent group.

The Linux kernel provides access to a series of controllers or subsystems for the cgroup technology. The controller is responsible for distributing a specific type of system resources to a set of one or more processes. For instance, the `memory` controller is what limits memory usage while the `cpuacct` controller monitors CPU usage.

You can access and manage cgroups both directly and indirectly (with LXC, libvirt or Docker), the first of which I cover here via sysfs and the `libcgroups` library. To follow along with the examples here, you first need to install the necessary packages. On Red Hat Enterprise Linux or CentOS, type the following on the command line:

```
$ sudo yum install libcgroup libcgroup-tools
```

On Ubuntu or Debian, type:

```
$ sudo apt-get install libcgroup1 cgroup-tools
```

For the example application, I'm using a simple shell script file called test.sh, and it'll be running the following two commands in an infinite `while` loop:

```
$ cat test.sh
#!/bin/sh

while [ 1 ]; do
        echo "hello world"
        sleep 60
done
```

The Manual Approach

With the proper packages installed, you can configure your cgroups directly via the sysfs hierarchy.

For instance, to create a cgroup named `foo` under the `memory` subsystem, create a directory named foo in /sys/fs/cgroup/memory:

```
$ sudo mkdir /sys/fs/cgroup/memory/foo
```

By default, every newly created cgroup will inherit access to the system's entire pool of memory. For some applications, primarily those that continue to allocate more memory but refuse to free what already has been allocated, that may not be such a great idea. To limit an application to a reasonable limit, you'll need to update the `memory.limit_in_bytes` file.

Limit the memory for anything running under the cgroup `foo` to 50MB:

```
$ echo 50000000 | sudo tee
 ↪/sys/fs/cgroup/memory/foo/memory.limit_in_bytes
```

Verify the setting:

```
$ sudo cat memory.limit_in_bytes
50003968
```

Note that the value read back always will be a multiple of the kernel's page size (that is, 4096 bytes or 4KB). This value is the smallest allocatable size of memory.

Launch the application:

```
$ sh ~/test.sh &
```

Using its Process ID (PID), move the application to cgroup `foo` under the `memory` controller:

```
$ echo 2845 > /sys/fs/cgroup/memory/foo/cgroup.procs
```

Using the same PID number, list the running process and verify that it's running within the desired cgroup:

```
$ ps -o cgroup 2845
CGROUP
8:memory:/foo,1:name=systemd:/user.slice/user-0.slice/
↪session-4.scope
```

You also can monitor what's currently being used by that cgroup by reading the desired files. In this case, you'll want to see the amount of memory allocated by your process (and spawned subprocesses):

```
$ cat /sys/fs/cgroup/memory/foo/memory.usage_in_bytes
253952
```

When a Process Goes Astray

Now let's re-create the same scenario, but instead of limiting the cgroup `foo` to 50MB of memory, you'll limit it to 500 bytes:

```
$ echo 500 | sudo tee /sys/fs/cgroup/memory/foo/
↪memory.limit_in_bytes
```

> ***NOTE:*** *if a task exceeds its defined limits, the kernel will intervene and, in some cases, kill that task.*

Again, when you read the value back, it always will be a multiple of the kernel page size. So, although you set it to 500 bytes, it's really set to 4 KB:

```
$ cat /sys/fs/cgroup/memory/foo/memory.limit_in_bytes
4096
```

Launch the application, move it into the cgroup and monitor the system logs:

```
$ sudo tail -f /var/log/messages
Oct 14 10:22:40 localhost kernel: sh invoked oom-killer:
 ↪gfp_mask=0xd0, order=0, oom_score_adj=0
Oct 14 10:22:40 localhost kernel: sh cpuset=/ mems_allowed=0
Oct 14 10:22:40 localhost kernel: CPU: 0 PID: 2687 Comm:
 ↪sh Tainted: G
OE  ------------   3.10.0-327.36.3.el7.x86_64 #1
Oct 14 10:22:40 localhost kernel: Hardware name: innotek GmbH
VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
Oct 14 10:22:40 localhost kernel: ffff880036ea5c00
 ↪0000000093314010 ffff88000002bcd0 ffffffff81636431
Oct 14 10:22:40 localhost kernel: ffff88000002bd60
 ↪ffffffff816313cc 01018800000000d0 ffff88000002bd68
Oct 14 10:22:40 localhost kernel: ffffffffbc35e040
 ↪fffeefff00000000 0000000000000001 ffff880036ea6103
Oct 14 10:22:40 localhost kernel: Call Trace:
Oct 14 10:22:40 localhost kernel: [<ffffffff81636431>]
 ↪dump_stack+0x19/0x1b
Oct 14 10:22:40 localhost kernel: [<ffffffff816313cc>]
 ↪dump_header+0x8e/0x214
```

```
Oct 14 10:22:40 localhost kernel: [<ffffffff8116d21e>]
 ↪oom_kill_process+0x24e/0x3b0
Oct 14 10:22:40 localhost kernel: [<ffffffff81088e4e>] ?
 ↪has_capability_noaudit+0x1e/0x30
Oct 14 10:22:40 localhost kernel: [<ffffffff811d4285>]
 ↪mem_cgroup_oom_synchronize+0x575/0x5a0
Oct 14 10:22:40 localhost kernel: [<ffffffff811d3650>] ?
 ↪mem_cgroup_charge_common+0xc0/0xc0
Oct 14 10:22:40 localhost kernel: [<ffffffff8116da94>]
 ↪pagefault_out_of_memory+0x14/0x90
Oct 14 10:22:40 localhost kernel: [<ffffffff8162f815>]
 ↪mm_fault_error+0x68/0x12b
Oct 14 10:22:40 localhost kernel: [<ffffffff816422d2>]
 ↪__do_page_fault+0x3e2/0x450
Oct 14 10:22:40 localhost kernel: [<ffffffff81642363>]
 ↪do_page_fault+0x23/0x80
Oct 14 10:22:40 localhost kernel: [<ffffffff8163e648>]
 ↪page_fault+0x28/0x30
Oct 14 10:22:40 localhost kernel: Task in /foo killed as
 ↪a result of limit of /foo
Oct 14 10:22:40 localhost kernel: memory: usage 4kB, limit
 ↪4kB, failcnt 8
Oct 14 10:22:40 localhost kernel: memory+swap: usage 4kB,
 ↪limit 9007199254740991kB, failcnt 0
Oct 14 10:22:40 localhost kernel: kmem: usage 0kB, limit
 ↪9007199254740991kB, failcnt 0
Oct 14 10:22:40 localhost kernel: Memory cgroup stats for /foo:
 ↪cache:0KB rss:4KB rss_huge:0KB mapped_file:0KB swap:0KB
 ↪inactive_anon:0KB active_anon:0KB inactive_file:0KB
 ↪active_file:0KB unevictable:0KB
Oct 14 10:22:40 localhost kernel: [ pid ]   uid  tgid total_vm
 ↪rss nr_ptes swapents oom_score_adj name
Oct 14 10:22:40 localhost kernel: [ 2687]     0  2687    28281
```

```
 ↪347     12        0             0 sh
Oct 14 10:22:40 localhost kernel: [ 2702]     0  2702    28281
 ↪50    7        0             0 sh
Oct 14 10:22:40 localhost kernel: Memory cgroup out of memory:
 ↪Kill process 2687 (sh) score 0 or sacrifice child
Oct 14 10:22:40 localhost kernel: Killed process 2702 (sh)
 ↪total-vm:113124kB, anon-rss:200kB, file-rss:0kB
Oct 14 10:22:41 localhost kernel: sh invoked oom-killer:
 ↪gfp_mask=0xd0, order=0, oom_score_adj=0
[ ... ]
```

Notice that the kernel's Out-Of-Memory Killer (or oom-killer) stepped in as soon as the application hit that 4KB limit. It killed the application, and it's no longer running. You can verify this by typing:

```
$ ps -o cgroup 2687
CGROUP
```

Using libcgroup

Many of the earlier steps described here are simplified by the administration utilities provided in the `libcgroup` package. For example, a single command invocation using the `cgcreate` binary takes care of the process of creating the sysfs entries and files.

To create the group named `foo` under the `memory` subsystem, type the following:

```
$ sudo cgcreate -g memory:foo
```

Using the same methods as before, you can begin to set thresholds:

```
$ echo 50000000 | sudo tee
 ↪/sys/fs/cgroup/memory/foo/memory.limit_in_bytes
```

> ***NOTE:*** *libcgroup provides a mechanism for managing tasks in control groups.*

Verify the newly configured setting:

```
$ sudo cat memory.limit_in_bytes
50003968
```

Run the application in the cgroup `foo` using the `cgexec` binary:

```
$ sudo cgexec -g memory:foo ~/test.sh
```

Using its PID number, verify that the application is running in the cgroup and under defined subsystem (`memory`):

```
$  ps -o cgroup 2945
CGROUP
6:memory:/foo,1:name=systemd:/user.slice/user-0.slice/
↪session-1.scope
```

If your application is no longer running and you want to clean up and remove the cgroup, you would do that by using the `cgdelete` binary. To remove group `foo` from under the `memory` controller, type:

```
$ sudo cgdelete memory:foo
```

Persistent Groups

You also can accomplish all of the above from a simple configuration file and the starting of a service. You can define all of your cgroup names and attributes in the /etc/cgconfig.conf file. The following appends a few attributes for the group `foo`:

```
$ cat /etc/cgconfig.conf
#
#  Copyright IBM Corporation. 2007
#
#  Authors:     Balbir Singh <balbir@linux.vnet.ibm.com>
#  This program is free software; you can redistribute it
#  and/or modify it under the terms of version 2.1 of the GNU
#  Lesser General Public License as published by the Free
#  Software Foundation.
#
#  This program is distributed in the hope that it would be
#  useful, but WITHOUT ANY WARRANTY; without even the implied
#  warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
#  PURPOSE.
#
#
# By default, we expect systemd mounts everything on boot,
# so there is not much to do.
# See man cgconfig.conf for further details, how to create
# groups on system boot using this file.

group foo {
  cpu {
    cpu.shares = 100;
  }
  memory {
    memory.limit_in_bytes = 5000000;
  }
}
```

The `cpu.shares` options defines the CPU priority of the group. By default, all groups inherit 1,024 shares or 100% of CPU time. By bringing this value down to something a bit more conservative, like 100, the group will be limited to approximately 10% of the CPU time.

As discussed earlier, a process running within a cgroup also can be limited to the amount of CPUs (cores) it can access. Add the following section to the same cgconfig.conf file and under the desired group name:

```
cpuset {
  cpuset.cpus="0-5";
}
```

With this limit, this cgroup will bind the application to cores 0 to 5—that is, it will see only the first six CPU cores on the system.

Next, you need to load this configuration using the `cgconfig` service. First, enable `cgconfig` to load the above configuration on system boot up:

```
$ sudo systemctl enable cgconfig
Create symlink from /etc/systemd/system/sysinit.target.wants/
↪cgconfig.service
to /usr/lib/systemd/system/cgconfig.service.
```

Now, start the `cgconfig` service and load the same configuration file manually (or you can skip this step and reboot the system):

```
$ sudo systemctl start cgconfig
```

Launch the application into the cgroup `foo` and bind it to your `memory` and `cpu` limits:

```
$ sudo cgexec -g memory,cpu,cpuset:foo ~/test.sh &
```

With the exception of launching the application into the predefined cgroup, all the rest will persist across system reboots. However, you can automate that process by defining a startup init script dependent on the `cgconfig` service to launch that same application.

Summary

Often it becomes necessary to limit one or more tasks on a machine. Control groups provide that functionality, and by leveraging it, you can enforce strict hardware and software limitations to some of your most vital or uncontrollable applications. If one application does not set an upper threshold or limit the amount of memory it can consume on a system, cgroups can address that. If another application tends to be a bit of a CPU hog, again, cgroups has got you covered. You can accomplish so much with cgroups, and with a little time invested, you'll restore stability, security and sanity back into your operating environment.

In Part II of this series, I move beyond Linux control groups and shift focus to how technologies like Linux Containers make use of them. ■

Petros Koutoupis, *LJ* Editor at Large, is currently a senior platform architect at IBM for its Cloud Object Storage division (formerly Cleversafe). He is also the creator and maintainer of the RapidDisk Project. Petros has worked in the data storage industry for well over a decade and has helped pioneer the many technologies unleashed in the wild today.

Resources

For other *LJ* articles on containers, see the following:

- "Make Your Containers Transparent with Puppet's Lumogon" by John S. Tonello
- "Concerning Containers' Connections: on Docker Networking" by Federico Kereki
- "Linux Containers and the Future Cloud" by Rami Rosen
- "Docker: Lightweight Linux Containers for Consistent Development and Deployment" by Dirk Merkel
- "Containers—Not Virtual Machines—Are the Future Cloud" by David Strauss

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

Everything You Need to Know about Linux Containers, Part II: Working with Linux Containers (LXC)

Part I of this Deep Dive on containers introduces the idea of kernel control groups, or cgroups, and the way you can isolate, limit and monitor selected userspace applications. Here, I dive a bit deeper and focus on the next step of process isolation—that is, through containers, and more specifically, the Linux Containers (LXC) framework.

By Petros Koutoupis

Containers are about as close to bare metal as you can get when running virtual machines. They impose very little to no overhead when hosting virtual instances. First introduced in 2008, LXC adopted much of its functionality from the Solaris Containers (or Solaris Zones) and FreeBSD jails that preceded it. Instead of creating a full-fledged virtual machine, LXC enables a virtual environment with its own process

What Exactly Are Containers?

The short answer is that containers decouple software applications from the operating system, giving users a clean and minimal Linux environment while running everything else in one or more isolated "containers". The purpose of a container is to launch a limited set of applications or services (often referred to as microservices) and have them run within a self-contained sandboxed environment.

and network space. Using namespaces to enforce process isolation and leveraging the kernel's very own control groups (cgroups) functionality, the feature limits, accounts for and isolates CPU, memory, disk I/O and network usage of one or more processes. Think of this userspace framework as a very advanced form of `chroot`.

This isolation prevents processes running within a given container from monitoring or affecting processes running in another container. Also, these containerized services do not influence or disturb the host machine. The idea of being able to consolidate

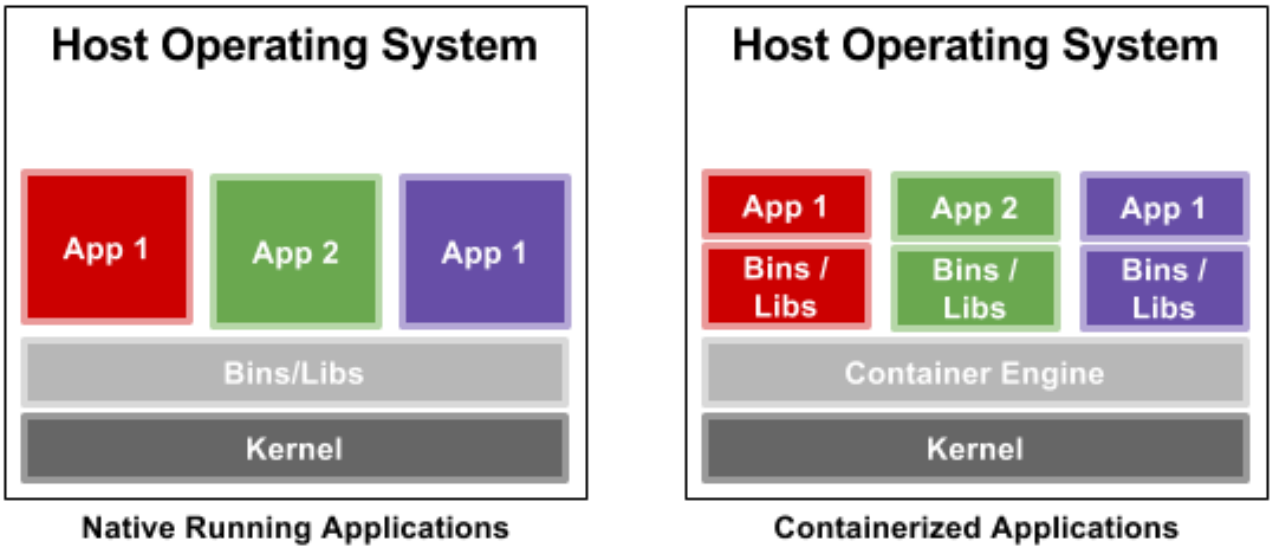


Figure 1. A Comparison of Applications Running in a Traditional Environment to Containers

many services scattered across multiple physical servers into one is one of the many reasons data centers have chosen to adopt the technology.

Container features include the following:

- Security: network services can be run in a container, which limits the damage caused by a security breach or violation. An intruder who successfully exploits a security hole on one of the applications running in that container is restricted to the set of actions possible within that container.

- Isolation: containers allow the deployment of one or more applications on the same physical machine, even if those applications must operate under different domains, each requiring exclusive access to its respective resources. For instance, multiple applications running in different containers can bind to the same physical network interface by using distinct IP addresses associated with each container.

- Virtualization and transparency: containers provide the system with a virtualized environment that can hide or limit the visibility of the physical devices or system's configuration underneath it. The general principle behind a container is to avoid changing the environment in which applications are running with the exception of addressing security or isolation issues.

Using the LXC Utilities

For most modern Linux distributions, the kernel is enabled with cgroups, but you most likely still will need to install the LXC utilities.

If you're using Red Hat or CentOS, you'll need to install the EPEL repositories first. For other distributions, such as Ubuntu or Debian, simply type:

```
$ sudo apt-get install lxc
```

Now, before you start tinkering with those utilities, you need to configure your environment. And before doing that, you need to verify that your current user has

both a `uid` and `gid` entry defined in /etc/subuid and /etc/subgid:

```
$ cat /etc/subuid
petros:100000:65536
$ cat /etc/subgid
petros:100000:65536
```

Create the ~/.config/lxc directory if it doesn't already exist, and copy the /etc/lxc/default.conf configuration file to ~/.config/lxc/default.conf. Append the following two lines to the end of the file:

```
lxc.id_map = u 0 100000 65536
lxc.id_map = g 0 100000 65536
```

It should look something like this:

```
$ cat ~/.config/lxc/default.conf
lxc.network.type = veth
lxc.network.link = lxcbr0
lxc.network.flags = up
lxc.network.hwaddr = 00:16:3e:xx:xx:xx
lxc.id_map = u 0 100000 65536
lxc.id_map = g 0 100000 65536
```

Append the following to the /etc/lxc/lxc-usernet file (replace the first column with your user name):

```
petros veth lxcbr0 10
```

The quickest way for these settings to take effect is either to reboot the node or log the user out and then log back in.

Once logged back in, verify that the `veth` networking driver is currently loaded:

```
$ lsmod|grep veth
veth                   16384  0
```

If it isn't, type:

```
$ sudo modprobe veth
```

You now can use the LXC utilities to download, run and manage Linux containers.

Next, download a container image and name it "example-container". When you type the following command, you'll see a long list of supported containers under many Linux distributions and versions:

```
$ sudo lxc-create -t download -n example-container
```

You'll be given three prompts to pick the distribution, release and architecture. I chose the following:

```
Distribution: ubuntu
Release: xenial
Architecture: amd64
```

Once you make a decision and press Enter, the rootfs will be downloaded locally and configured. For security reasons, each container does not ship with an OpenSSH server or user accounts. A default root password also is not provided. In order to change the root password and log in, you must run either an `lxc-attach` or `chroot` into the container directory path (after it has been started).

Start the container:

```
$ sudo lxc-start -n example-container -d
```

The `-d` option dæmonizes the container, and it will run in the background. If you want

to observe the boot process, replace the `-d` with `-F`, and it will run in the foreground, ending at a login prompt.

You may experience an error similar to the following:

```
$ sudo lxc-start -n example-container -d
lxc-start: tools/lxc_start.c: main: 366 The container
failed to start.
lxc-start: tools/lxc_start.c: main: 368 To get more details,
run the container in foreground mode.
lxc-start: tools/lxc_start.c: main: 370 Additional information
can be obtained by setting the --logfile and --logpriority
options.
```

If you do, you'll need to debug it by running the `lxc-start` service in the foreground:

```
$ sudo lxc-start -n example-container -F
lxc-start: conf.c: instantiate_veth: 2685 failed to create veth
 pair (vethQ4NS0B and vethJMHON2): Operation not supported
    lxc-start: conf.c: lxc_create_network: 3029 failed to
    create netdev
    lxc-start: start.c: lxc_spawn: 1103 Failed to create
    the network.
    lxc-start: start.c: __lxc_start: 1358 Failed to spawn
    container "example-container".
    lxc-start: tools/lxc_start.c: main: 366 The container failed
    to start.
    lxc-start: tools/lxc_start.c: main: 370 Additional information
    can be obtained by setting the --logfile and --logpriority
    options.
```

From the example above, you can see that the `veth` module probably isn't inserted. After inserting it, it resolved the issue.

Anyway, open up a second terminal window and verify the status of the container:

```
$ sudo lxc-info -n example-container
Name:           example-container
State:          RUNNING
PID:            1356
IP:             10.0.3.28
CPU use:        0.29 seconds
BlkIO use:      16.80 MiB
Memory use:     29.02 MiB
KMem use:       0 bytes
Link:           vethPRK7YU
 TX bytes:      1.34 KiB
 RX bytes:      2.09 KiB
 Total bytes:   3.43 KiB
```

Another way to do this is by running the following command to list all installed containers:

```
$ sudo lxc-ls -f
NAME         STATE   AUTOSTART GROUPS IPV4       IPV6
example-container RUNNING 0         -      10.0.3.28 -
```

But there's a problem; you still can't log in! Attach directly to the running container, create your users and change all relevant passwords using the `passwd` command:

```
$ sudo lxc-attach -n example-container
root@example-container:/#
root@example-container:/# useradd petros
root@example-container:/# passwd petros
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

Once the passwords are changed, you'll be able to log in directly to the container from a console and without the `lxc-attach` command:

```
$ sudo lxc-console -n example-container
```

If you want to connect to this running container over the network, install the OpenSSH server:

```
root@example-container:/# apt-get install openssh-server
```

Grab the container's local IP address:

```
root@example-container:/# ip addr show eth0|grep inet
    inet 10.0.3.25/24 brd 10.0.3.255 scope global eth0
    inet6 fe80::216:3eff:fed8:53b4/64 scope link
```

Then from the host machine and in a new console window, type:

```
$ ssh 10.0.3.25
```

Voilà! You now can `ssh` in to the running container and type your user name and password.

On the host system, and not within the container, it's interesting to observe which LXC processes are initiated and running after launching a container:

```
$ ps aux|grep lxc|grep -v grep
root       861  0.0  0.0 234772  1368 ?        Ssl  11:01
 ↪0:00 /usr/bin/lxcfs /var/lib/lxcfs/
lxc-dns+  1155  0.0  0.1  52868  2908 ?        S    11:01
 ↪0:00 dnsmasq -u lxc-dnsmasq --strict-order
 ↪--bind-interfaces --pid-file=/run/lxc/dnsmasq.pid
 ↪--listen-address 10.0.3.1 --dhcp-range 10.0.3.2,10.0.3.254
```

```
 ↪--dhcp-lease-max=253 --dhcp-no-override
 ↪--except-interface=lo --interface=lxcbr0
 ↪--dhcp-leasefile=/var/lib/misc/dnsmasq.lxcbr0.leases
 ↪--dhcp-authoritative
root      1196  0.0  0.1  54484  3928 ?        Ss   11:01
 ↪0:00 [lxc monitor] /var/lib/lxc example-container
root      1658  0.0  0.1  54780  3960 pts/1    S+   11:02
 ↪0:00 sudo lxc-attach -n example-container
root      1660  0.0  0.2  54464  4900 pts/1    S+   11:02
 ↪0:00 lxc-attach -n example-container
```

To stop a container, type (from the host machine):

```
$ sudo lxc-stop -n example-container
```

Once stopped, verify the state of the container:

```
$ sudo lxc-ls -f
NAME         STATE   AUTOSTART GROUPS IPV4 IPV6
example-container STOPPED 0         -      -    -

$ sudo lxc-info -n example-container
Name:           example-container
State:          STOPPED
```

To destroy a container completely—that is, purge it from the host system—type:

```
$ sudo lxc-destroy -n example-container
Destroyed container example-container
```

Once destroyed, verify that it has been removed:

```
$ sudo lxc-info -n example-container
```

```
example-container doesn't exist

$ sudo lxc-ls -f
$
```

Note: if you attempt to destroy a running container, the command will fail and inform you that the container is still running:

```
$ sudo lxc-destroy -n example-container
example-container is running
```

A container must be stopped before it is destroyed.

Advanced Configurations

At times, it may be necessary to configure one or more containers to accomplish one or more tasks. LXC simplifies this by having the administrator modify the container's configuration file located in /var/lib/lxc:

```
$ sudo su
# cd /var/lib/lxc
# ls
example-container
```

The container's parent directory will consist of at least two files: 1) the container config file and 2) the container's entire rootfs:

```
# cd example-container/
# ls
config  rootfs
```

Say you want to autostart the container labeled example-container on host system boot up. To do this, you'd need to append the following lines to the container's configuration file, /var/lib/lxc/example-container/config:

```
# Enable autostart
lxc.start.auto = 1
```

After you restart the container or reboot the host system, you should see something like this:

```
$ sudo lxc-ls -f
NAME              STATE   AUTOSTART GROUPS IPV4       IPV6
example-container RUNNING 1         -      10.0.3.25 -
```

Notice how the `AUTOSTART` field is now set to "1".

If, on container boot up, you want the container to bind mount a directory path living on the host machine, append the following lines to the same file:

```
# Bind mount system path to local path
lxc.mount.entry = /mnt mnt none bind 0 0
```

With the above example and when the container gets restarted, you'll see the contents of the host's /mnt directory accessible to the container's local /mnt directory.

Privileged vs. Unprivileged Containers

You often may stumble across LXC-related content discussing the idea of a privileged container and an unprivileged container. But what are those exactly? The concept is pretty straightforward, and an LXC container can run in either configuration.

By design, an unprivileged container is considered safer and more secure than a privileged one. An unprivileged container runs with a mapping of the container's root UID to a non-root UID on the host system. This makes it more difficult for attackers compromising a container to gain root privileges to the underlying host machine. In short, if attackers manage to compromise your container through, for example, a known software vulnerability, they immediately will find themselves with no rights on the host machine.

Privileged containers can and will expose a system to such attacks. That's why it's good practice to run few containers in privileged mode. Identify the containers that require privileged access, and be sure to make extra efforts to update routinely and lock them down in other ways.

And, What about Docker?

I spent a considerable amount of time talking about Linux Containers, but what about Docker? It *is* the most deployed container solution in production. Since its initial launch, Docker has taken the Linux computing world by storm. Docker is an Apache-licensed open-source containerization technology designed to automate the repetitive task of creating and deploying microservices inside containers. Docker treats containers as if they were extremely lightweight and modular virtual machines. Initially, Docker was built on top of LXC, but it has since moved away from that dependency, resulting in a better developer and user experience. Much like LXC, Docker continues to make use of the kernel cgroup subsystem. The technology is more than just running containers, it also eases the process of creating containers, building images, sharing those built images and versioning them.

Docker primarily focuses on:

- Portability: Docker provides an image-based deployment model. This type of portability allows for an easier way to share an application or set of services (with all of their dependencies) across multiple environments.

- Version control: a single Docker image is made up of a series of combined layers. A new layer is created whenever the image is altered. For instance, a new layer is created every time a user specifies a command, such as `run` or `copy`. Docker will reuse these layers for new container builds. Layering to Docker is its very own method of version control.

- Rollback: again, every Docker image has layers. If you don't want to use the currently running layer, you can roll back to a previous version. This type of agility makes it easier for software developers to integrate and deploy their software technology continuously.

- Rapid deployment: provisioning new hardware often can take days. And, the amount of effort and overhead to get it installed and configured is quite burdensome. With Docker, you can avoid all of that by reducing the time it takes to get an image up and running to a matter of seconds. When you're done with a container, you can destroy it just as easily.

Fundamentally, both Docker and LXC are very similar. They both are userspace and lightweight virtualization platforms that implement cgroups and namespaces to manage resource isolation. However, there are a number of distinct differences between the two.

Process Management Docker restricts containers to run as a single process. If your application consists of X number of concurrent processes, Docker will want you to run X number of containers, each with its own distinct process. This is not the case with LXC, which runs a container with a conventional init process and, in turn, can host multiple processes inside that same container. For example, if you want to host a LAMP (Linux + Apache + MySQL + PHP) server, each process for each application will need to span across multiple Docker containers.

State Management Docker is designed to be stateless, meaning it doesn't support persistent storage. There are ways around this, but again, it's only necessary when the process requires it. When a Docker image is created, it will consist of read-only layers. This will not change. During runtime, if the process of the container makes any changes to its internal state, a diff between the internal state and the current state of the image will be maintained until either a commit is made to the Docker image (creating a new layer) or until the container is deleted, resulting in that diff disappearing.

Portability This word tends to be overused when discussing Docker—that's because it's the single-most important advantage Docker has over LXC. Docker does a much better job of abstracting away the networking, storage and operating system details from the application. This results in a truly configuration-independent application, guaranteeing that the environment for the application always will remain the same, regardless of the machine on which it is enabled.

Docker is designed to benefit both developers and system administrators. It has made itself an integral part of many DevOps (developers + operations) toolchains. Developers can focus on writing code without having to worry about the system ultimately hosting it. With Docker, there's no need to install and configure complex databases or worry about switching between incompatible language toolchain versions. Docker gives the operations staff flexibility, often reducing the number of physical systems needed to host some of the smaller and more basic applications. Docker streamlines software delivery. New features and bug/security fixes reach the customer quickly without any hassle, surprises or downtime.

Summary

Isolating processes for the sake of infrastructure security and system stability isn't as painful as it sounds. The Linux kernel provides all the necessary facilities to enable simple-to-use userspace applications, such as LXC (and even Docker), to manage micro-instances of an operating system with its local services in an isolated and sandboxed environment.

In Part III of this series, I describe container orchestration with Kubernetes. ■

Petros Koutoupis, *LJ* Editor at Large, is currently a senior platform architect at IBM for its Cloud Object Storage division (formerly Cleversafe). He is also the creator and maintainer of the RapidDisk Project. Petros has worked in the data storage industry for well over a decade and has helped pioneer the many technologies unleashed in the wild today.

Resources

- Linux Containers
- Docker vs LXC
- Debian Wiki: LXC
- LXC Ubuntu Documentation
- GitHub LXC

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

Everything You Need to Know about Containers, Part III: Orchestration with Kubernetes

A look at using Kubernetes to create, deploy and manage thousands of container images.

By Petros Koutoupis

If you've read the first two articles in this series, you now should be familiar with Linux kernel control groups, Linux Containers and Docker. But, here's a quick recap: once upon a time, data-center administrators deployed entire operating systems, occupying entire hardware servers to host a few applications each. This was a lot of overhead with a lot to manage. Now scale that across multiple server hosts, and it increasingly became more difficult to maintain. This was a problem—a problem that wasn't easily solved. It would take time for technological evolution to reach the moment where you are able to shrink the operating system and launch these varied applications as microservices hosted across multiple containers on the same physical machine.

In the final part of this series, I explore the method most people use to create, deploy and manage containers. The concept is typically referred to as container orchestration. If I were to focus on Docker, on its own, the technology is extremely

simple to use, and running a few images simultaneously is also just as easy. Now, scale that out to hundreds, if not thousands, of images. How do you manage that? Eventually, you need to step back and rely on one of the few orchestration frameworks specifically designed to handle this problem. Enter Kubernetes.

Kubernetes

Kubernetes, or k8s (k + eight characters), originally was developed by Google. It's an open-source platform aiming to automate container operations: "deployment, scaling and operations of application containers across clusters of hosts". Google was an early adopter and contributor to the Linux Container technology (in fact, Linux Containers power Google's very own cloud services). Kubernetes eliminates all of the manual processes involved in the deployment and scaling of containerized applications. It's capable of clustering together groups of servers hosting Linux Containers while also allowing administrators to manage those clusters easily and efficiently.

Kubernetes makes it possible to respond to consumer demands quickly by deploying your applications within a timely manner, scaling those same applications with ease and seamlessly rolling out new features, all while limiting hardware resource consumption. It's extremely modular and can be hooked into by other applications or frameworks easily. It also provides additional self-healing services, including auto-placement, auto-replication and auto-restart of containers.

There also exists Docker's own platform called Swarm. It accomplishes much of the same tasks and boasts a lot of the same features. The primary difference between the two is that Swarm is centralized around the use of Docker, while Kubernetes tends to adopt a more generalized container support model.

Sometimes production applications will span across multiple containers, and those containers may be deployed across multiple physical server machines. Both Kubernetes and Swarm give you the orchestration and management capabilities required to deploy and scale those containers to accommodate the always changing workload requirements.

Architecture

Kubernetes runs on top of an operating system (such as Ubuntu Server, Red Hat Enterprise Linux, SUSE Linux Enterprise Server and so on) and takes a master-slave approach to its functionality. The *master* signifies the machine (physical or virtual) that controls the Kubernetes nodes. This is where all tasks originate. It *is* the main controlling unit of the cluster and will take the commands issued by an administrator or DevOps team and, in turn, relay them to the underlying nodes. The master node can be configured to run on a single machine or across multiple machines in a high-availability cluster. This is to ensure fault-tolerance of the cluster and reduce the likelihood of downtime. The *nodes* are the machines that perform the tasks assigned by the master. The node is sometimes referred to as a Worker or Minion.

Kubernetes is broken down into a set of components, some of which manage individual nodes, while the rest are part of the control plane.

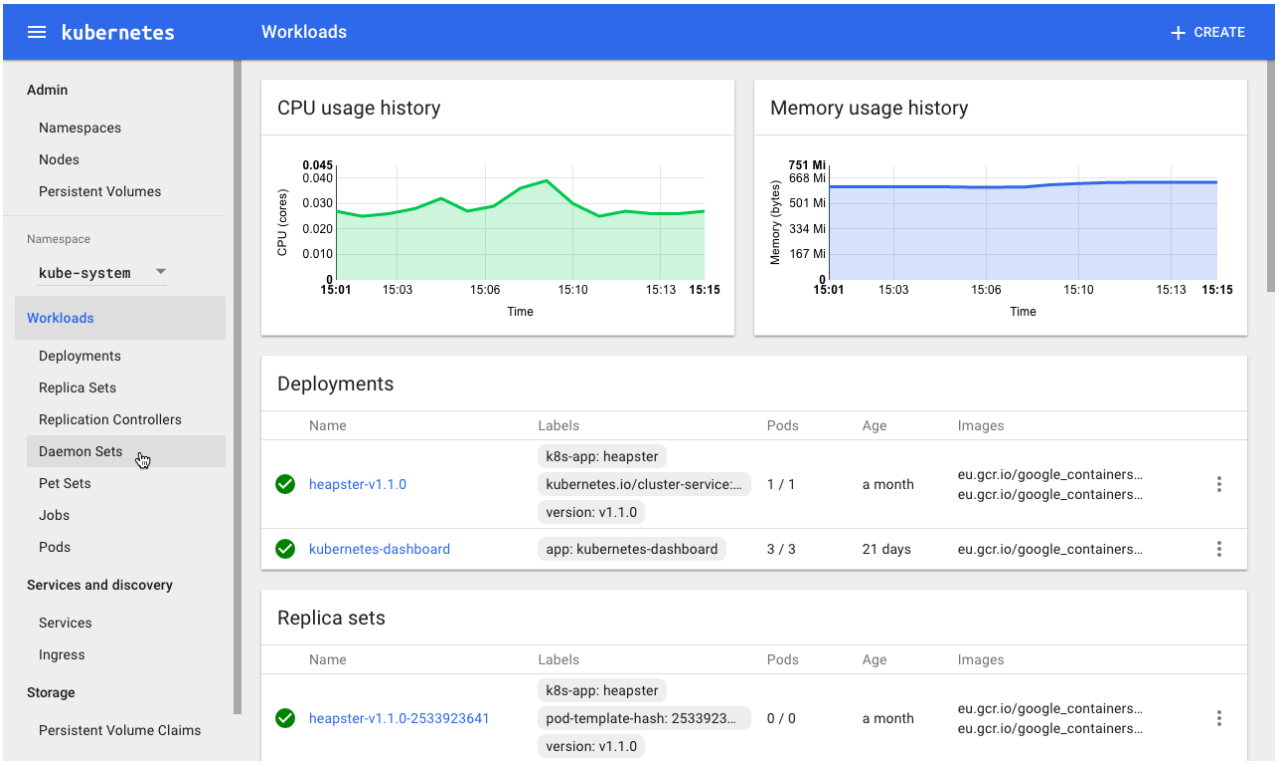


Figure 1. The Kubernetes Web UI Dashboard (Source: kubernetes.io)

Control plane management:

- etcd: is a lightweight and distributed cluster manager. It's persistent and reliably stores the configuration data of the cluster, providing a consistent and accurate representation of the cluster at any given point of time.

- API server: serves the Kubernetes API using JSON over HTTP. It provides both an internal and external interface to Kubernetes. The server processes and validates RESTful requests and enables communication between and across several tools and libraries.

- Scheduler: selects on which node an unscheduled pod should run. This logic is based on resource availability. The scheduler also tracks resource utilization of each node, ensuring that the assigned workload never exceeds what is available on the physical or virtual machine.

- Control Manager: the process hosting the DaemonSet and Replication controllers. The controllers communicate with the API server to create, update or delete managed resources.

Node management:

- kubelet: responsible for the running state of each node and making sure that all containers on the node are healthy. It handles the starting and stopping of application containers (see how this differs with Docker in the next section) within a pod as directed by the manager in the control plane.

- kube-proxy: a network proxy and load balancer. It's responsible for routing traffic to the appropriate container.

- cAdvisor: an agent that monitors and collects system resource utilization and performance metrics (such as CPU, memory, file and network) of each container on each node.

Controllers A controller drives the state of the cluster by managing a set of pods. The *Replication Controller* handles pod replication and scaling by running a specified number of copies of a given pod across the entire cluster of nodes. It also can handle the creation of replacement pods in the event of a failing node. The *DaemonSet Controller* is in charge of running exactly one pod per node. The *Job Controller* runs pods to completion (that is, as part of a batch job).

Services In Kubernetes terms, a *service* consists of a set of pods working together (a one-tier or multi-tier application). As Kubernetes provides service discovery and request routing (by assigning the appropriate static networking parameters), it ensures that all service requests get to the right pod, regardless of where it moves across the cluster. Some of this movement may be a result of pod or node failure. In the end, Kubernetes' self-healing capabilities will get those ailing services back to a pristine state automatically.

Pods When a Kubernetes master deploys a group of one or more containers to a single node, it does so by creating a *pod*. Pods abstract the networking and storage from the container, and all of the containers within a pod will share the same IP address, hostname

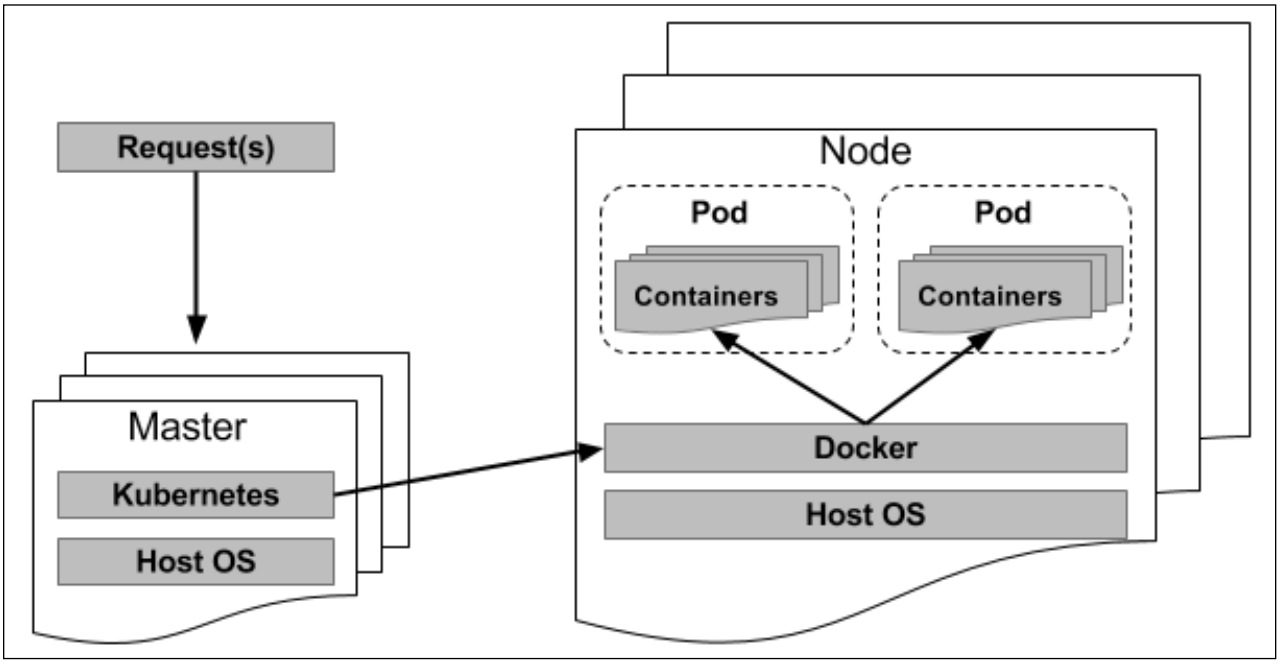


Figure 2. A General Model of Pod Creation/Management

and more, allowing it to be moved around in the cluster without complication.

The kubelet will monitor each and every pod. If it's not in a good state, it will redeploy that pod to the same node. Apart from this, a heartbeat messaging mechanism will relay the node status to the master every few seconds. As soon as the master detects a node failure, the Replication Controller will launch the now affected pods onto another healthy node.

So, how does Docker fit into all of this? Docker still functions as it was meant to function. When a Kubernetes master schedules a pod to a node, the kubelet running on that node will direct Docker in launching the desired containers. The kubelet will continue by monitoring those containers while also collecting information for the master. Docker still will be in full control of the containers running on the node and also will be responsible for starting and stopping them. The only difference here is that you now have an automated system sending these requests to Docker instead of the system administrator running the same tasks manually.

Spinning Up a Kubernetes Installation

Modern Linux distributions have made the installation and configuration of a Kubernetes host quite simple. I use Ubuntu Server 16.04 for the following example. *Note: you'll need a substantial amount of memory and storage to run with this example properly.*

To begin, install `conjure-up`:

```
$ sudo snap install conjure-up --classic
```

`conjure-up` is a neat wrapper around Juju, MAAS and LXD. It's advertised as a turnkey solution to enable big and complicated software stacks—Kubernetes included. `conjure-up` essentially processes a collection of scripts leveraging the previously named technologies.

Next, install `lxd`:

```
$ sudo snap install lxd
```

LXD is Canonical's (Ubuntu's) homegrown container technology. Whereas Docker is more focused on deploying applications, LXD specializes in deploying Linux virtual machines.

In order to meet all requirements for installation to the localhost, you'll need to create at least one LXD storage pool:

```
$ sudo /snap/bin/lxc storage create kube-test dir source=/mnt
Storage pool kube-test created
```

You can view the newly created pool with the following command:

```
$ sudo /snap/bin/lxc storage list
+-----------+-------------+--------+--------+---------+
|   NAME    | DESCRIPTION | DRIVER | SOURCE | USED BY |
+-----------+-------------+--------+--------+---------+
| kube-test |             | dir    | /mnt   | 0       |
+-----------+-------------+--------+--------+---------+
```

You'll also need to create a networking bridge:

```
$ /snap/bin/lxc network create lxdbr0 ipv4.address=auto
 ↪ipv4.nat=true ipv6.address=none ipv6.nat=false

Network lxdbr0 created
```

Run `conjure-up`:

```
$ conjure-up
```

You'll see a menu, where you can select the Canonical distribution of Kubernetes.

Then you'll be prompted with the option to install various and useful add-on packages to your Kubernetes deployment.

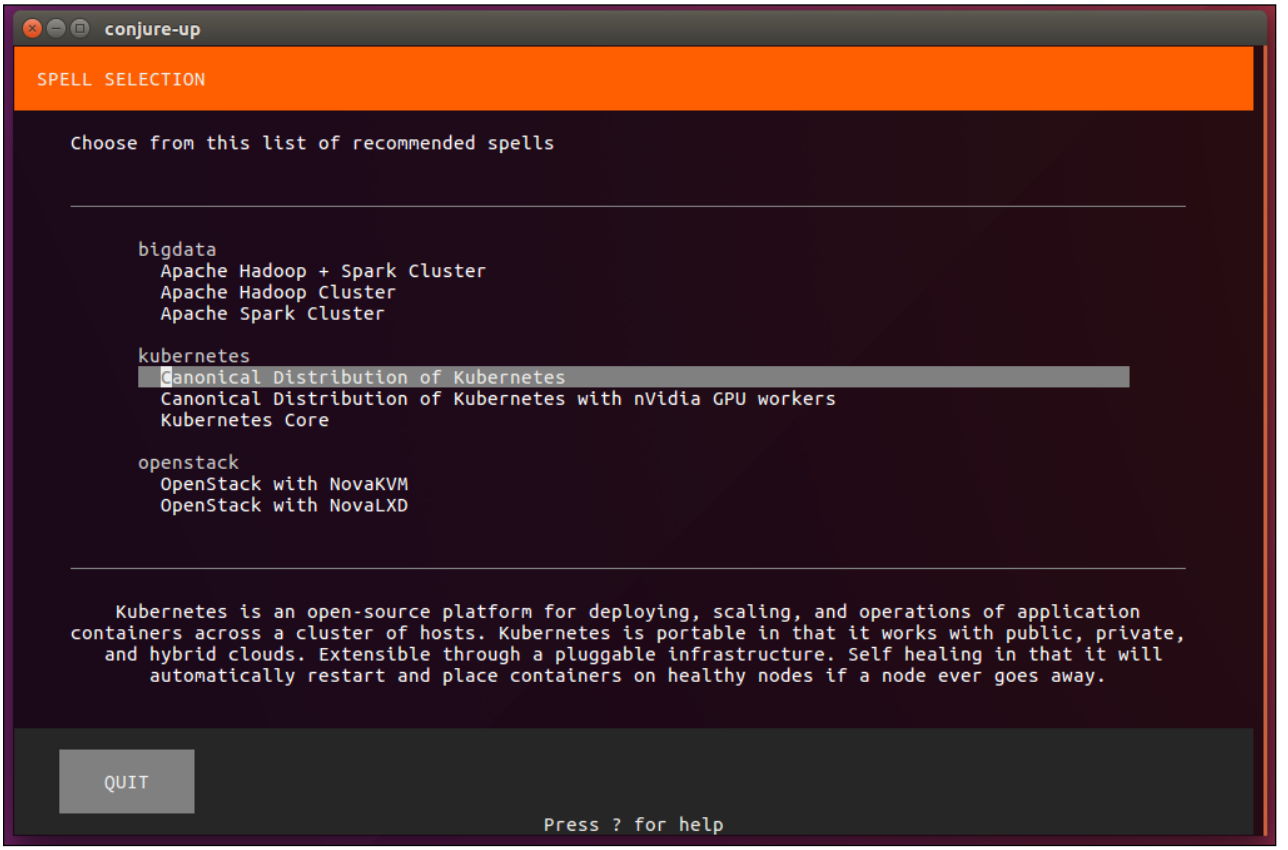


Figure 3. The `conjure-up` Framework Selection Menu

A few more options will be presented, such as where to deploy (for example, the cloud or locally). For the purpose of this example, let's install and deploy to the localhost, so in the following menu, select the network bridge you created earlier (`lxdbr0`) and the storage pool (`kube-test`). A couple simple questions later, and the installation process begins. The entire process will take quite a while.

Hopefully, installing the Kubernetes main components doesn't take too long, but let's assume that by this point, everything is completed. As part of the post-install process, the kubectl client application will be installed, and then the host system will capture the Kubernetes cluster status. You will be provided with an installation summary.

When everything has completed, run the following `kubectl` command:

```
$ ~/kubectl cluster-info
```

You'll see a short display of the cluster's running components, including addresses to various dashboards and services.

Cloud Native Computing

Cloud native computing, often referred to as serverless computing, is not only the latest trending buzzword in the data center, but it also offers a new way of hosting applications. The idea challenges what traditionally has been the norm and puts more power into the application itself while abstracting away everything underneath it. But before getting into the details of serverless computing, here's a crash course in cloud computing.

Going Serverless Cloud native computing is a relatively recent term describing the more-modern trend of deploying and managing applications. The idea is pretty straightforward. Each application or process is packaged into its own container, which in turn is dynamically orchestrated (that is, scheduled and managed) across a cluster of nodes. This approach moves applications away from physical hardware and operating system dependency and into their own self-contained and sandboxed environment, which can run anywhere within the data center transparently and seamlessly. The cloud native approach is about separating the various components of application delivery.

The Evolution of Containers in the Cloud As you would expect, container technology has helped accelerate cloud adoption. Think about it. You have these persistent containerized application images that within seconds are spun up or down as needed and balanced across multiple nodes or data-center locations to achieve the best in quality of service (QoS). Even the big-time public cloud providers make use of the same container technologies and for the same reason: rapid application deployment. For instance, Amazon, Microsoft and Google provide their container services with Docker. And as it applies to the greater serverless ecosystem, the applications hosted in those containers are stateless and event-triggered. This means that a third-party component will manage access to this application, as it is needed and invoked.

Now, when I think of a true serverless solution, one of the first things that comes

to mind is Amazon's AWS Lambda. Amazon takes serverless to the next level with Lambda by spinning up a container to host the applications you need, ensuring access and availability for your business or service. Under this model, there is no need to provision or manage physical or virtual servers. Assuming it is in a stable or production state, you just deploy your code, and you're done. With Lambda, you don't manage the container (further reducing your overhead). Your code is just deployed within an isolated containerized environment. It's pretty straightforward. AWS Lambda enables user-defined code functions to be triggered directly via a user-defined HTTPS request. The way Lambda differs from traditional containerized deployment is that Amazon has provided a framework for developers to upload their event-driven application code (written in Node.js, Python, Java or C#) and respond to events, such as website clicks, within milliseconds. All libraries and dependencies to run the bulk of your code are provided for within the container. Lambda scales automatically to support the exact needs of your application.

As for the types of events (labeled an event source) on which to trigger your application, or code handlers, Amazon has made it so you can trigger on website visits or clicks, a REST HTTP request to its API gateway, a sensor reading on your Internet of Things (IoT) device, or even an upload of a photograph to an S3 bucket. This API gateway forms the bridge that connects all parts of AWS Lambda. For example, a developer can write a handler to trigger on HTTPS request events.

Let's say you need to enable a level of granularity to your code. Lambda accommodates this by allowing developers to write modular handlers. For instance, you can write one handler to trigger for each API method, and each handler can be invoked, updated and altered independently of the others.

Lambda allows developers to combine all required dependencies (that is, libraries, native binaries or even external web services) to your function into a single package, giving a handler the freedom to reach out to any of those dependencies as it needs them.

Now, how does this compare to an Amazon AWS Elastic Cloud Computing (EC2) instance? Well, the short answer is that it's a lot more simplified, and by simplified, I

The Cloud Native Computing Foundation

Formed in 2015, the Cloud Native Computing Foundation (CNCF) was assembled to help standardize these recent paradigm shifts in hosting Cloud services—that is, to unify and define the cloud native era. Although the primary goal of the foundation is to be the best place to host cloud native software projects. The foundation is home to many cloud-centric projects, including the Kubernetes orchestration framework.

To help standardize this new trend of computing, the foundation has divided the entire architecture into a set of subsystems, each with its own set of standardized APIs for inter-component communication. Subsystems include orchestration, resource scheduling and distributed systems services.

You can learn more about the foundation by visiting the foundation's official website.

mean that there's zero to no overhead on configuring or maintaining your operating environment. If you need more out of your environment that requires access to a full-blown operating system or container, you can spin up an EC2 virtual instance. EC2 provides users the flexibility to customize their virtual machine with both the hardware and software it will host. If you only need to host a function or special-purpose application, that's where Lambda becomes the better choice. With Lambda, there isn't much to customize—and sometimes, less is good.

Summary

Kubernetes expands beyond the management of the traditional container and allows you to scale to meet consumer demands effectively and efficiently. And with modern and major Linux distributions, deploying a Kubernetes cluster is as simple as running a script and answering a few questions.

As you explore this wonderful technology further, know that you are not alone. There are companies that provide services and solutions centered around Kubernetes. One such company is Heptio, which was founded by Kubernetes co-creators Craig McLuckie and Joe Beda. Centered around both developers and system administrators, Heptio's products and services simplify and scale the Kubernetes ecosystem.

There is also the need to maintain both security and compliance of your container images within that same ecosystem. Again, when you scale to the thousands, management of such things is near impossible. That's where companies like Twistlock do the heavy-lifting for you. Twistlock develops and distributes a product of the same name focusing on nothing but Docker image security and compliance. It also can be operated from and managed by orchestration platforms including Kubernetes. ■

Petros Koutoupis, *LJ* Editor at Large, is currently a senior platform architect at IBM for its Cloud Object Storage division (formerly Cleversafe). He is also the creator and maintainer of the RapidDisk Project. Petros has worked in the data storage industry for well over a decade and has helped pioneer the many technologies unleashed in the wild today.

Resources

Kubernetes main website is here.

Further reading on Kubernetes from *Linux Journal*:

- "Kubernetes, Four Years Later, and Amazon Redefining Container Orchestration" by Petros Koutoupis
- "AWS Quick Start for Kubernetes" by Craig McLuckie
- Joe Beda, Co-Founder and CTO of Heptio, on Becoming a Cloud Native Organization
- "An Interview with Heptio, the Kubernetes Pioneers" by Petros Koutoupis

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

The Search for a GUI Docker

Docker is everything but pretty; let's try to fix that. Here's a rundown of some GUI options available for Docker.

By Shawn Powers

I love Docker. At first it seemed a bit silly to me for a small-scale implementation like my home setup, but after learning how to use it, I fell in love. The standard features are certainly beneficial. It's great not worrying that one application's dependencies will step on or conflict with another's. But most applications are good about playing well with others, and package management systems keep things in order. So why do I `docker run` instead of `apt-get install`? Individualized system settings.

With Docker, I can have three of the same apps running side by side. They even can use the same port (internally) and not conflict. My torrent client can live inside a forced-VPN network, and I don't need to worry that it will somehow "leak" my personal IP data. Heck, I can run apps that work only on CentOS inside my Ubuntu Docker server, and it just works! In short, Docker is amazing.

I just wish I could remember all the commands.

Don't get me wrong, I'm familiar with Docker. I use it for most of my server needs. It's my first go-to when testing a new app. Heck, I taught an entire course on Docker for CBT Nuggets (my day job). The problem is, Docker works so well, I rarely need to interact with it. So, my FIFO buffer fills up, and I forget the simple command-line options to make Docker work. Also, because I like charts and graphs, I decided to install a Docker GUI. It was a bit of an adventure, so I thought I'd share the ins and outs of my experience.

My GUI Expectations

There are some things I don't really care about for a GUI. Oddly, one of the most common uses people have for a visual interface is the ability to create a Docker container. I actually don't mind using the command line when I'm creating a container, because it usually takes 5–10 attempts and tweaks before I get it how I want it. So for me, I'd like to have at least the following features:

- A visual layout of all containers, whether or not they're running.
- A way to start/stop/delete containers.
- The ability to rename running containers, because I always forget to name them, and I get tired of seeing "chubby_cheetah" for container names.
- A way to change the restart policy easily, so when I finally get a container right, I can have it `--restart=always`.
- Show some statistics about the system and individual containers.
- Read logs.
- Work via web interface, so I can use it remotely.
- Be a Docker container itself!

My list of needs is fairly simple, but oddly, many GUIs left me wanting. Since everyone's desires are different, I'll go over the most popular options I tried, and mention some pros and cons.

Kitematic

It has a strange name, but Kitematic has been around for a very long time. It actually was adopted officially by Docker, and it's now part of the Docker Toolkit. It's a fairly simple interface, but simplicity is sometimes a bonus, because finding the things it

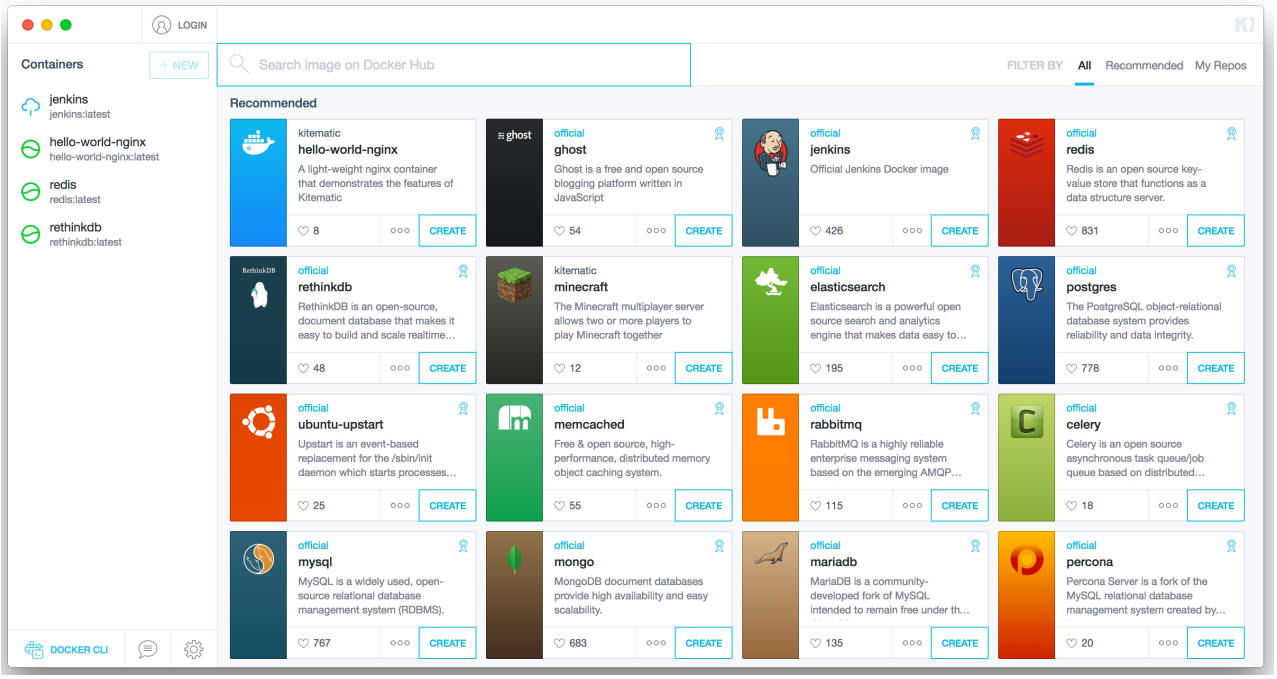


Figure 1. Kitematic looks incredibly nice, but it's unfortunately a native non-web app (image via kitematic.com).

can do is straightforward.

If you check out the Kitematic website, you might get the impression it runs only on OS X or Windows, but thankfully, that's not the case. If you head over to the GitHub repository, you'll find an Ubuntu installer as well.

Kitematic is pretty, but it's a native application, not a web interface. It looks nice (Figure 1), but I'm often on someone else's computer when I'm out and about, so my web-based requirement is really vital. Still, if you don't need a web interface and want a supported, clean GUI, Kitematic is pretty slick. It's fairly simple, but again, that's not a bad thing.

Pros:

- Officially supported by Docker.
- Very clear and simple interface.

Cons:

- Not a web app.

DockStation

DockStation is another non-web-based solution, but I wanted to mention it because it's very powerful. It has a great, clean interface (Figure 2), but again, it requires installation on a local machine. It does have support for accessing a remote Docker server over SSH, but that still means installing an application locally, and configuration isn't straightforward.

Plus, DockStation is closed source. That alone wouldn't be a showstopper for me, but when added to the frustration of a native application install, it's enough to turn me off. Still, it is free for personal use, and especially if you're a developer, DockStation might be very attractive. There are lots of screenshots on the website if you want to check it out.

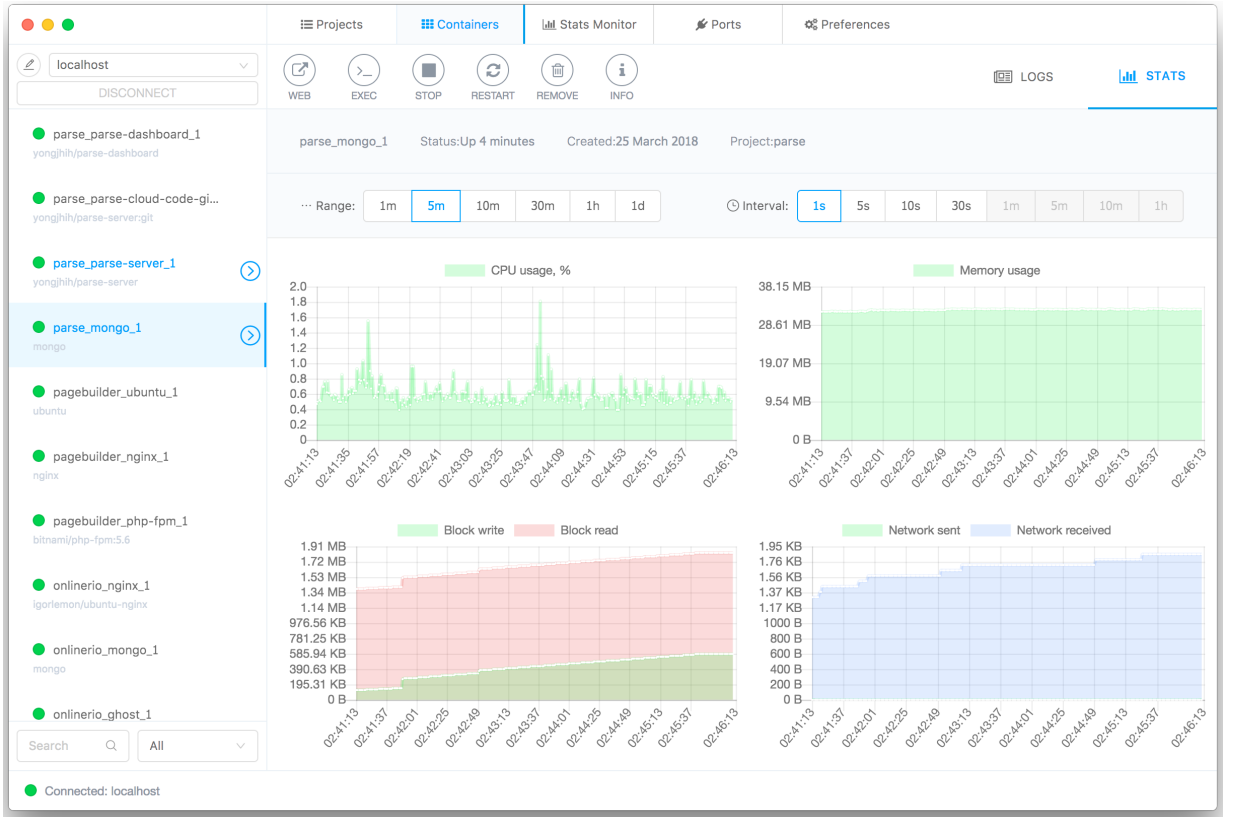


Figure 2. DockStation also looks great, but it's proprietary and requires quite a bit of work to get working. Also, it's not web-based (image via dockstation.io).

Pros:

- Well-designed interface.
- More powerful than Kitematic and has more features.
- Supports remote SSH access (but is complex to configure).

Cons:

- Not open source.
- Not web-based.

Portainer

The closest thing I found to a perfect solution was Portainer. It's web-based, runs as a Docker container, and it's easy to get going. On the surface, it seems fairly basic, but I was impressed to see there is even a tiered user system, meaning you can give various users different levels of access to the Docker infrastructure.

Portainer doesn't seem to keep historical performance statistics, but it does have a real-time monitor (Figure 3) that works in a pinch if I'm trying to troubleshoot a misbehaving container. I'm a little surprised to find that the application template feature (Figure 4) is one of my favorite things to play with. I know my original requirements did not include a way to deploy containers quickly, but grabbing a quick template and clicking deploy is surprisingly simple, and I find myself playing with containers more often. It's not that deploying a container via the command line is hard, but it's a lot of typing; whereas with Portainer, I simply can click "deploy", and it's all done for me. It seems silly, but anything that gets me playing with technology is a good thing.

If you want a fairly simple interface to your running Docker server, Portainer is great. It can start and stop containers, edit existing running apps (and even rename them, something I always have to do), and it's a great way to clean up your system after a session of adding/removing images. For example, you can look at the volumes tab (Figure 5) and quickly identify volumes no longer associated with a running container. It's something I never think about, so I had hundreds of abandoned volumes just sitting there.

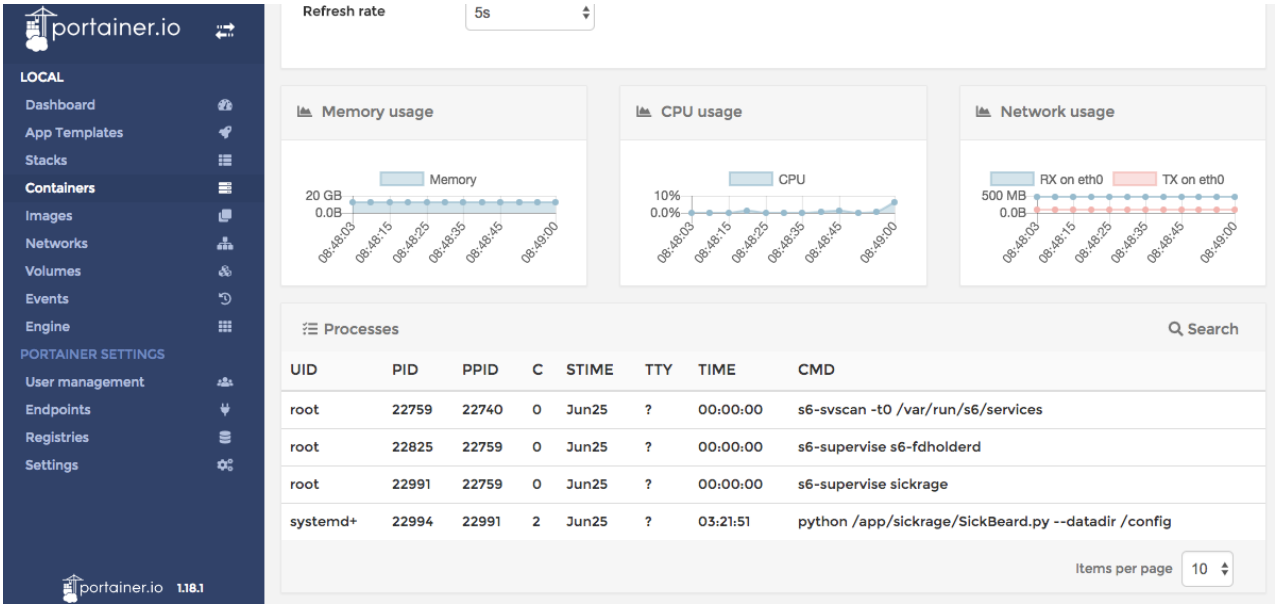


Figure 3. I love seeing graphs, and if a container is behaving poorly, this real-time look at performance is very useful.

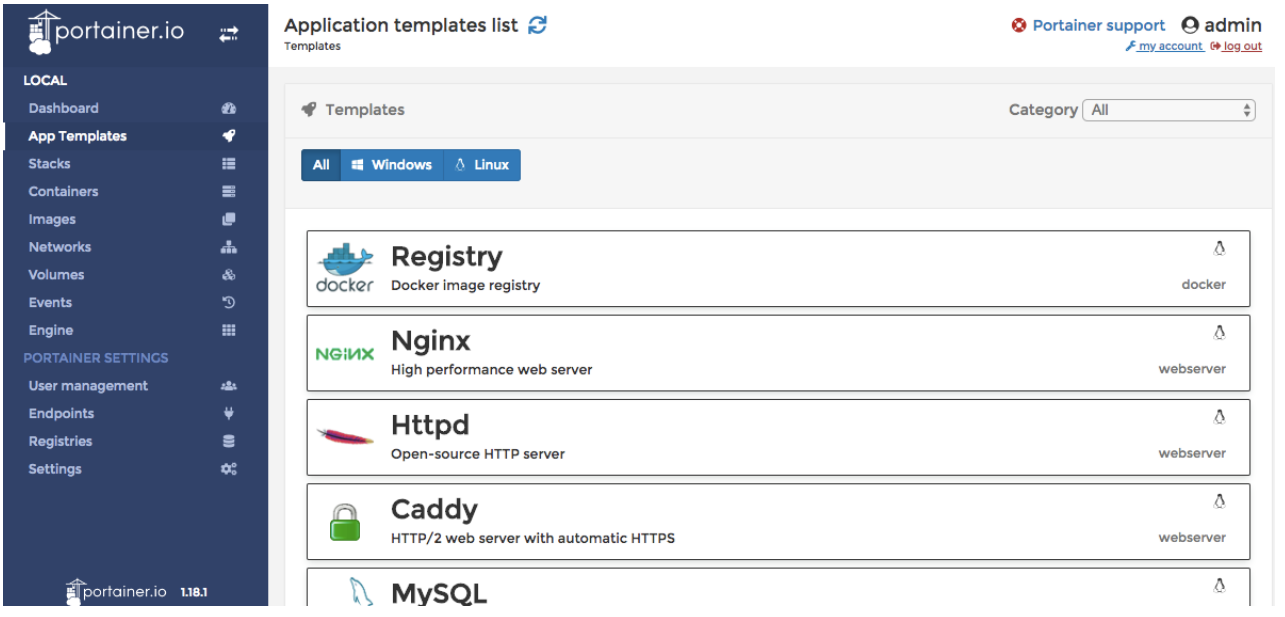


Figure 4. I'm not normally into templates like this, but I kind of felt like a kid in a candy store here!

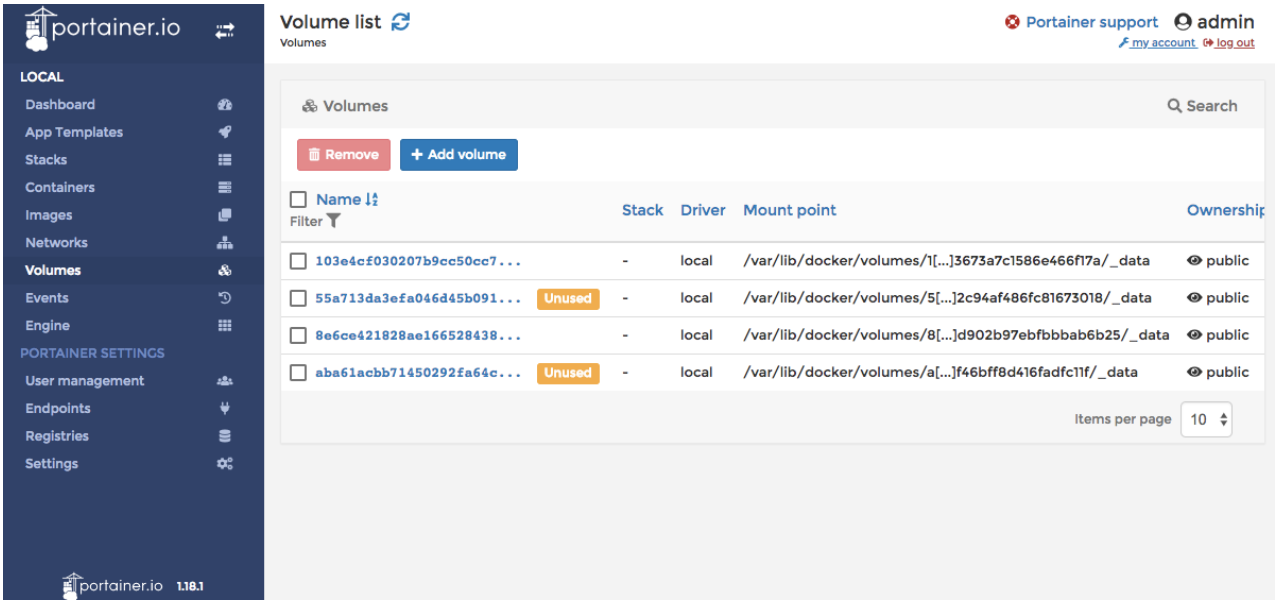


Figure 5. It's not that abandoned volumes were taking up a lot of room on my system, but I hate leaving things lying around. Portainer made cleanup easy.

My go-to system for interacting with running containers is Portainer. It does what I need in a pinch, runs via web browser, and it has enough features that it's worth keeping around. It's open source, and it's being actively developed. I highly recommend it. In fact, one of the best things about Portainer is how easy it is to set up. The following one-liner will download the image, get the container going and give you access to the interface on port 9000:

```
docker run -d -p 9000:9000 \
-v /var/run/docker.sock:/var/run/docker.sock \
portainer/portainer
```

I don't expose that to the internet, but when I'm out and about, a quick `ssh` command with a port-forward allows me full access to the Portainer instance. Thanks to the UNIX socket system, Portainer directly interacts with Docker, and there aren't any system configuration changes to make in order to get things done. That one-liner above is literally all it takes. (And for me, the first order of business in the GUI

is to change the name of the container, because I always forget to do that when creating one!)

The folks at Portainer.io offer a fully functional demo here (login is admin:tryportainer), which will give you a feel for what it can do without deploying it in your own environment. I'll admit, I've had mixed luck getting the demo to work (it often hangs on me), so it might be easier to spin up a quick container and check it out on your own system. Still, offering a demo is nice.

Pros:

- Web-based.
- Simple setup (it runs in a container).
- Easy to navigate.
- User management.

Cons:

- Couldn't figure out how to change the restart policy on containers!

Dockly, the Weird Cousin

I'd be remiss not to mention one more interesting option, which is technically a GUI, although not in the traditional sense. Dockly is a GUI, but it's a Curses-based text GUI. Basically, if Kyle Rankin were looking for a GUI interface to Docker, Dockly likely would be his first choice. Heck, it even looks a bit like Mutt (Figure 6).

Admittedly, navigating the GUI via the keyboard is a bit cumbersome at first. It took me a while to get the feel for what Dockly really can do. Once I got used to the interface, however, I must admit it's pretty nice. Of all the management systems, it's definitely the easiest to connect with, because it runs in an SSH window.

For instructions on how to set up and use Dockly, head over to the GitHub page. It uses node.js, which has been an issue for me in the past regarding conflicting versions. But since this is Docker, it can, of course, be run in its own container,

making sure it has the exact version it needs.

Pros:

- Runs from a container.
- Easy to set up.
- Kyle Rankin-friendly (terminal window only).

Cons:

- Limited ability to modify containers.

My Final Thoughts

You know how sometimes you search the internet for the name of a particular actor, and then three hours later find yourself watching YouTube videos of cats getting scared by cucumbers? Well, my dive in to the world of Docker GUI interfaces was something like that. Portainer was one of the first I tried, and it ended up being the

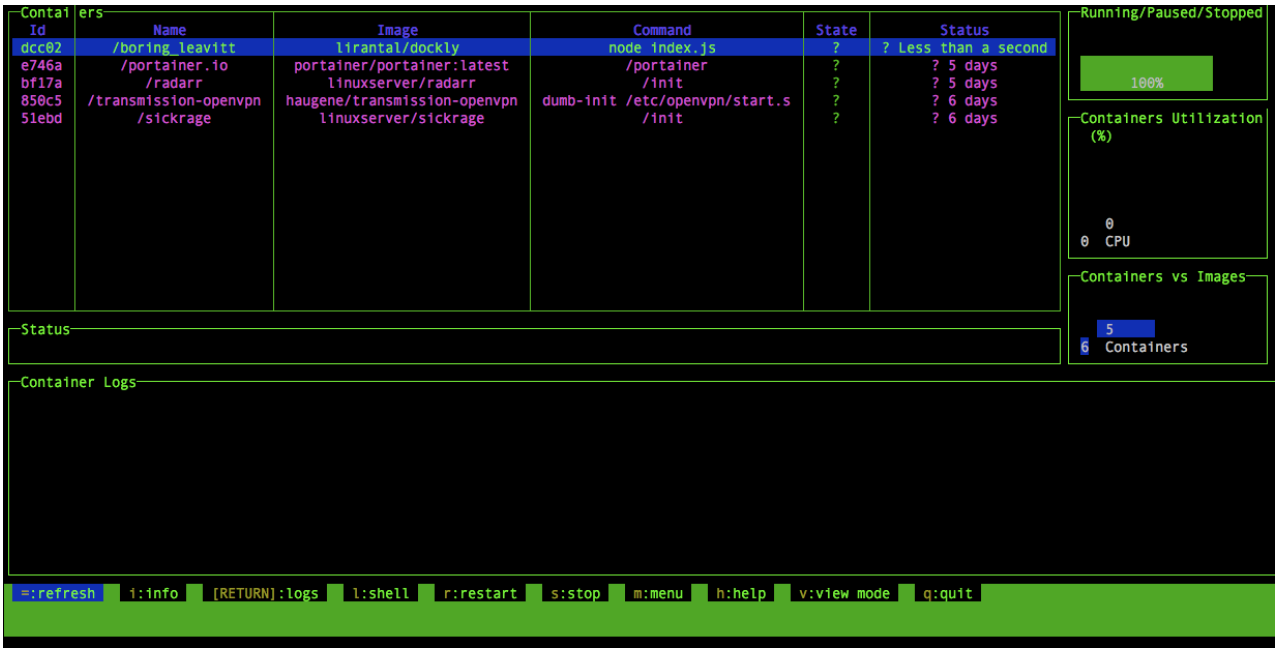


Figure 6. Dockly isn't the prettiest GUI for Docker, but it does most of the same things the other GUIs do, and it functions completely inside a terminal window. It's worth a look.

best. Still, I spent more than five hours installing and configuring various GUIs, ranging from the now defunct "Shipyard" to the extremely advanced "Rancher". Many were native applications, several required additional VMs installed locally in order to run, and most were more complicated than I wanted.

I honestly wish Kitematic was a web-based GUI, because its integration with Docker is nice, and its simplistic look makes it easy to manage a simple infrastructure. If you're at the same computer all the time, checking out a native-GUI option might be worth your time. In the end, even though I have Portainer and Dockly running, I actually find myself `ssh`-ing into my Docker server and running commands by hand. It can be cumbersome, and I don't always remember the particular arguments. Still, it's just as easy to google for the answer as it is to log in to a GUI.

But the graphs—the graphs are awful nice to look at! ■

Shawn is Associate Editor here at *Linux Journal*, and has been around Linux since the beginning. He has a passion for open source, and he loves to teach. He also drinks too much coffee, which often shows in his writing. You can contact Shawn via e-mail, ljeditor@linuxjournal.com.

Sharing Docker Containers across DevOps Environments

Docker provides a powerful tool for creating lightweight images and containerized processes, but did you know it can make your development environment part of the DevOps pipeline too? Whether you're managing tens of thousands of servers in the cloud or are a software engineer looking to incorporate Docker containers into the software development life cycle, this article has a little something for everyone with a passion for Linux and Docker.

By Todd A. Jacobs

In this article, I describe how Docker containers flow through the DevOps pipeline. I also cover some advanced DevOps concepts (borrowed from object-oriented programming) on how to use dependency injection and encapsulation to improve the DevOps process. And finally, I show how containerization can be useful for the development and testing process itself, rather than just as a place to serve up an application after it's written.

Introduction

Containers are hot in DevOps shops, and their benefits from an operations and service delivery point of view have been covered well elsewhere. If you want to build a Docker container or deploy a Docker host, container or swarm, a lot of information is available. However, very few articles talk about how to *develop* inside the Docker containers that will be reused later in the DevOps pipeline, so that's what I focus on here.

Container-Based Development Workflows Two common workflows exist for developing software for use inside Docker containers:

1. Injecting development tools into an existing Docker container: this is the best option for sharing a consistent development environment with the same toolchain among multiple developers, and it can be used in conjunction with web-based development environments, such as Red Hat's codenvy.com or dockerized IDEs like Eclipse Che.

2. Bind-mounting a host directory onto the Docker container and using your existing development tools on the host: this is the simplest option, and it offers flexibility for developers to work with their own set of locally installed development tools.

Both workflows have advantages, but local mounting is inherently simpler. For that reason, I focus on the mounting solution as "the simplest thing that could possibly work" here.

How Docker Containers Move between Environments A core tenet of DevOps is that the source code and runtimes that will be used in production

Figure 1. Stages a Docker Container Moves Through in a Typical DevOps Pipeline

are the same as those used in development. In other words, the most effective pipeline is one where the identical Docker image can be reused for each stage of the pipeline.

The notion here is that each environment uses the same Docker image and code base, regardless of where it's running. Unlike systems such as Puppet, Chef or Ansible that converge systems to a defined state, an idealized Docker pipeline makes duplicate copies (containers) of a fixed image in each environment. Ideally, the only artifact that really moves between environmental stages in a Docker-centric pipeline is the ID of a Docker image; all other artifacts should be shared between environments to ensure consistency.

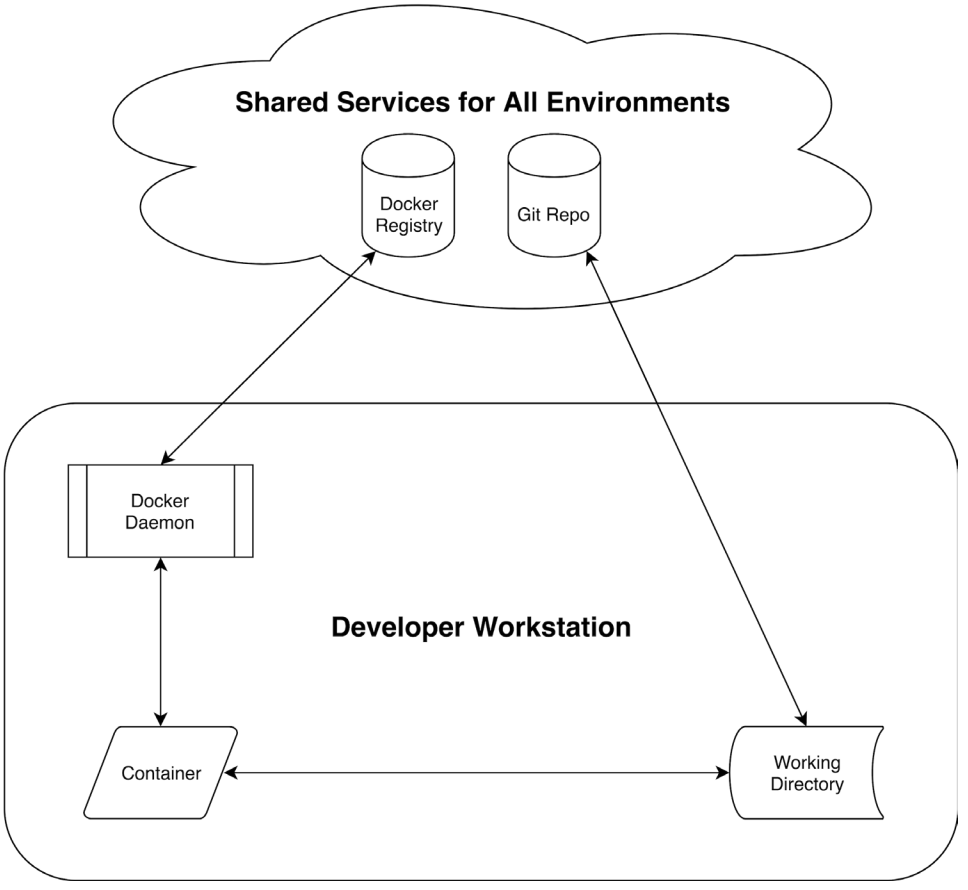


Figure 2. Idealized Docker-Based DevOps Pipeline

Handling Differences between Environments In the real world, environmental stages can vary. As a case point, your QA and staging environments may contain different DNS names, different firewall rules and almost certainly different data fixtures. Combat this per-environment drift by standardizing services across your different environments. For example, ensuring that DNS resolves "db1.example.com" and "db2.example.com" to the right IP addresses in each environment is much more Docker-friendly than relying on configuration file changes or injectable templates that point your application to differing IP addresses. However, when necessary, you can set environment variables for each container rather than making stateful changes to the fixed image. These variables then can be managed in a variety of ways, including the following:

1. Environment variables set at container runtime from the command line.

2. Environment variables set at container runtime from a file.

3. Autodiscovery using etcd, Consul, Vault or similar.

Consider a Ruby microservice that runs inside a Docker container. The service accesses a database somewhere. In order to run the same Ruby image in each different environment, but with environment-specific data passed in as variables, your deployment orchestration tool might use a shell script like this one, "Example Mircoservice Deployment":

```
# Reuse the same image to create containers in each
# environment.
docker pull ruby:latest

# Bash function that exports key environment
# variables to the container, and then runs Ruby
# inside the container to display the relevant
# values.
microservice () {
```

```
    docker run -e STAGE -e DB --rm ruby \
        /usr/local/bin/ruby -e \
            'printf("STAGE: %s, DB: %s\n",
                    ENV["STAGE"],
                    ENV["DB"])'
}
```

Table 1 shows an example of how environment-specific information for Development, Quality Assurance and Production can be passed to otherwise-identical containers using exported environment variables.

Table 1. Same Image with Injected Environment Variables

DEVELOPMENT	QUALITY ASSURANCE	PRODUCTION
`export STAGE=dev DB=db1; microservice`	`export STAGE=qa DB=db2; microservice`	`export STAGE=prod DB=db3; microservice`

To see this in action, open a terminal with a Bash prompt and run the commands from the "Example Microservice Deployment" script above to pull the Ruby image onto your Docker host and create a reusable shell function. Next, run each of the commands from the table above in turn to set up the proper environment variables and execute the function. You should see the output shown in Table 2 for each simulated environment.

Table 2. Containers in Each Environment Producing Appropriate Results

DEVELOPMENT	QUALITY ASSURANCE	PRODUCTION
`STAGE: dev, DB: db1`	`STAGE: qa, DB: db2`	`STAGE: prod, DB: db3`

Despite being a rather simplistic example, what's being accomplished is really quite extraordinary! This is DevOps tooling at its best: you're re-using the same image and deployment script to ensure maximum consistency, but each deployed instance (a "container" in Docker parlance) is still being tuned to operate properly within its

pipeline stage.

With this approach, you limit configuration drift and variance by ensuring that the exact same image is re-used for each stage of the pipeline. Furthermore, each container varies only by the environment-specific data or artifacts injected into them, reducing the burden of maintaining multiple versions or per-environment architectures.

But What about External Systems?

The previous simulation didn't really connect to any services outside the Docker container. How well would this work if you needed to connect your containers to environment-specific things outside the container itself?

Next, I simulate a Docker container moving from development through other stages of the DevOps pipeline, using a different database with its own data in each environment. This requires a little prep work first.

First, create a workspace for the example files. You can do this by cloning the examples from GitHub or by making a directory. As an example:

```
# Clone the examples from GitHub.
git clone \
    https://github.com/CodeGnome/SDCAPS-Examples
cd SDCAPS-Examples/db

# Create a working directory yourself.
mkdir -p SDCAPS-Examples/db
cd SDCAPS-Examples/db
```

The following SQL files should be in the db directory if you cloned the example repository. Otherwise, go ahead and create them now.

db1.sql:

```
-- Development Database
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE AppData (
    login TEXT UNIQUE NOT NULL,
    name TEXT,
    password TEXT
);
INSERT INTO AppData
VALUES ('root','developers','dev_password'),
       ('dev','developers','dev_password');
COMMIT;
```

db2.sql:

```
-- Quality Assurance (QA) Database
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE AppData (
    login TEXT UNIQUE NOT NULL,
    name TEXT,
    password TEXT
);
INSERT INTO AppData
VALUES ('root','qa admins','admin_password'),
       ('test','qa testers','user_password');
COMMIT;
```

db3.sql:

```
-- Production Database
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
```

```
CREATE TABLE AppData (
    login TEXT UNIQUE NOT NULL,
    name TEXT,
    password TEXT
);
INSERT INTO AppData
VALUES ('root','production',
        '$1$Ax6DIG/K$TDPdujixy5DDscpTWD5HU0'),
       ('deploy','devops deploy tools',
        '$1$hgTsycNO$FmJInHWROtkX6q7eWiJ1p/');
COMMIT;
```

Next, you need a small utility to create (or re-create) the various SQLite databases. This is really just a convenience script, so if you prefer to initialize or load the SQL by hand or with another tool, go right ahead:

```
#!/usr/bin/env bash

# You assume the database files will be stored in an
# immediate subdirectory named "db" but you can
# override this using an environment variable.
: "${DATABASE_DIR:=db}"
cd "$DATABASE_DIR"

# Scan for the -f flag. If the flag is found, and if
# there are matching filenames, verbosely remove the
# existing database files.
pattern='(^|[[:space:]])-f([[:space:]]|$)'
if [[ "$*" =~ $pattern ]] &&
    compgen -o filenames -G 'db?' >&-
then
    echo "Removing existing database files ..."
    rm -v db? 2> /dev/null
```

```
    echo
fi

# Process each SQL dump in the current directory.
echo "Creating database files from SQL ..."
for sql_dump in *.sql; do
    db_filename="${sql_dump%%.sql}"
    if [[ ! -f "$db_filename" ]]; then
        sqlite3 "$db_filename" < "$sql_dump" &&
        echo "$db_filename created"
    else
        echo "$db_filename already exists"
    fi
done
```

When you run ./create_databases.sh, you should see:

```
Creating database files from SQL ...
db1 created
db2 created
db3 created
```

If the utility script reports that the database files already exist, or if you want to reset the database files to their initial state, you can call the script again with the `-f` flag to re-create them from the associated .sql files.

Creating a Linux Password

You probably noticed that some of the SQL files contained clear-text passwords while others have valid Linux password hashes. For the purposes of this article, that's largely a contrivance to ensure that you have different data in each database and to make it easy to tell which database you're looking at from the data itself.

For security though, it's usually best to ensure that you have a properly hashed

password in any source files you may store. There are a number of ways to generate such passwords, but the OpenSSL library makes it easy to generate salted and hashed passwords from the command line.

Tip: for optimum security, don't include your desired password or passphrase as an argument to OpenSSL on the command line, as it could then be seen in the process list. Instead, allow OpenSSL to prompt you with `Password:` *and be sure to use a strong passphrase.*

To generate a salted MD5 password with OpenSSL:

```
$ openssl passwd \
    -1 \
    -salt "$(openssl rand -base64 6)"
Password:
```

Then you can paste the salted hash into /etc/shadow, an SQL file, utility script or wherever else you may need it.

Simulating Deployment inside the Development Stage

Now that you have some external resources to experiment with, you're ready to simulate a deployment. Let's start by running a container in your development environment. I follow some DevOps best practices here and use fixed image IDs and defined gem versions.

DevOps Best Practices for Docker Image IDs To ensure that you're re-using the same image across pipeline stages, always use an image ID rather than a named tag or symbolic reference when pulling images. For example, while the "latest" tag might point to different versions of a Docker image over time, the SHA-256 identifier of an image version remains constant and also provides automatic validation as a checksum for downloaded images.

Furthermore, you always should use a fixed ID for assets you're injecting into your

containers. Note how you specify a specific version of the SQLite3 Ruby gem to inject into the container at each stage. This ensures that each pipeline stage has the same version, regardless of whether the most current version of the gem from a RubyGems repository changes between one container deployment and the next.

Getting a Docker Image ID When you pull a Docker image, such as `ruby:latest`, Docker will report the digest of the image on standard output:

```
$ docker pull ruby:latest
latest: Pulling from library/ruby
Digest:
sha256:eed291437be80359321bf66a842d4d542a789e
↪687b38c31bd1659065b2906778
Status: Image is up to date for ruby:latest
```

If you want to find the ID for an image you've already pulled, you can use the `inspect` sub-command to extract the digest from Docker's JSON output—for example:

```
$ docker inspect \
      --format='{{index .RepoDigests 0}}' \
      ruby:latest
      ruby@sha256:eed291437be80359321bf66a842d4d542a789
↪e687b38c31bd1659065b2906778
```

First, you export the appropriate environment variables for development. These values will override the defaults set by your deployment script and affect the behavior of your sample application:

```
# Export values we want accessible inside the Docker
# container.
export STAGE="dev" DB="db1"
```

Next, implement a script called container_deploy.sh that will simulate

deployment across multiple environments. This is an example of the work that your deployment pipeline or orchestration engine should do when instantiating containers for each stage:

```
#!/usr/bin/env bash

set -e

######################################################
# Default shell and environment variables.
######################################################
# Quick hack to build the 64-character image ID
# (which is really a SHA-256 hash) within a
# magazine's line-length limitations.
hash_segments=(
    "eed291437be80359321bf66a842d4d54"
    "2a789e687b38c31bd1659065b2906778"
)
printf -v id "%s" "${hash_segments[@]}"

# Default Ruby image ID to use if not overridden
# from the script's environment.
: "${IMAGE_ID:=$id}"

# Fixed version of the SQLite3 gem.
: "${SQLITE3_VERSION:=1.3.13}"

# Default pipeline stage (e.g. dev, qa, prod).
: "${STAGE:=dev}"

# Default database to use (e.g. db1, db2, db3).
: "${DB:=db1}"
```

```
# Export values that should be visible inside the
# container.
export STAGE DB

#####################################################
# Setup and run Docker container.
#####################################################
# Remove the Ruby container when script exits,
# regardless of exit status unless DEBUG is set.
cleanup () {
    local id msg1 msg2 msg3
    id="$container_id"
    if [[ ! -v DEBUG ]]; then
        docker rm --force "$id" >&-
    else
        msg1="DEBUG was set."
        msg2="Debug the container with:"
        msg3="    docker exec -it $id bash"
        printf "\n%s\n%s\n%s\n" \
          "$msg1" \
          "$msg2" \
          "$msg3" \
          > /dev/stderr
  fi
}
trap "cleanup" EXIT

# Set up a container, including environment
# variables and volumes mounted from the local host.
docker run \
    -d \
    -e STAGE \
    -e DB \
```

```
    -v "${DATABASE_DIR:-${PWD}/db}":/srv/db \
    --init \
    "ruby@sha256:$IMAGE_ID" \
    tail -f /dev/null >&-

# Capture the container ID of the last container
# started.
container_id=$(docker ps -ql)

# Inject a fixed version of the database gem into
# the running container.
echo "Injecting gem into container..."
docker exec "$container_id" \
    gem install sqlite3 -v "$SQLITE3_VERSION" &&
    echo

# Define a Ruby script to run inside our container.
#
# The script will output the environment variables
# we've set, and then display contents of the
# database defined in the DB environment variable.
ruby_script='
    require "sqlite3"

    puts %Q(DevOps pipeline stage: #{ENV["STAGE"]})
    puts %Q(Database for this stage: #{ENV["DB"]})
    puts
    puts "Data stored in this database:"

    Dir.chdir "/srv/db"
    db    = SQLite3::Database.open ENV["DB"]
    query = "SELECT rowid, * FROM AppData"
    db.execute(query) do |row|
```

```
        print " " * 4
        puts row.join(", ")
    end
'

# Execute the Ruby script inside the running
# container.
docker exec "$container_id" ruby -e "$ruby_script"
```

There are a few things to note about this script. First and foremost, your real-world needs may be either simpler or more complex than this script provides for. Nevertheless, it provides a reasonable baseline on which you can build.

Second, you may have noticed the use of the `tail` command when creating the Docker container. This is a common trick used for building containers that don't have a long-running application to keep the container in a running state. Because you are re-entering the container using multiple `exec` commands, and because your example Ruby application runs once and exits, `tail` sidesteps a lot of ugly hacks needed to restart the container continually or keep it running while debugging.

Go ahead and run the script now. You should see the same output as listed below:

```
$ ./container_deploy.sh
Building native extensions.  This could take a while...
Successfully installed sqlite3-1.3.13
1 gem installed

DevOps pipeline stage: dev
Database for this stage: db1

Data stored in this database:
    1, root, developers, dev_password
    2, dev, developers, dev_password
```

Simulating Deployment across Environments Now you're ready to move on to something more ambitious. In the preceding example, you deployed a container to the development environment. The Ruby application running inside the container used the development database. The power of this approach is that the exact same process can be re-used for each pipeline stage, and the only thing you need to change is the database to which the application points.

In actual usage, your DevOps configuration management or orchestration engine would handle setting up the correct environment variables for each stage of the pipeline. To simulate deployment to multiple environments, populate an associative array in Bash with the values each stage will need and then run the script in a `for` loop:

```
declare -A env_db
env_db=([dev]=db1 [qa]=db2 [prod]=db3)

for env in dev qa prod; do
    export STAGE="$env" DB="${env_db[$env]}"
    printf "%s\n" "Deploying to ${env^^} ..."
    ./container_deploy.sh
done
```

This stage-specific approach has a number of benefits from a DevOps point of view. That's because:

1. The image ID deployed is identical across all pipeline stages.

2. A more complex application can "do the right thing" based on the value of `STAGE` and `DB` (or other values) injected into the container at runtime.

3. The container is connected to the host filesystem the same way at each stage, so you can re-use source code or versioned artifacts pulled from Git, Nexus or other repositories without making changes to the image or container.

4. The switcheroo magic for pointing to the right external resources is handled by your deployment script (in this case, container_deploy.sh) rather than by making changes to your image, application or infrastructure.

5. This solution is great if your goal is to trap most of the complexity in your deployment tools or pipeline orchestration engine. However, a small refinement would allow you to push the remaining complexity onto the pipeline infrastructure instead.

Imagine for a moment that you have a more complex application than the one you've been working with here. Maybe your QA or staging environments have large data sets that you don't want to re-create on local hosts, or maybe you need to point at a network resource that may move around at runtime. You can handle this by using a well known name that is resolved by a external resource instead.

You can show this at the filesystem level by using a symlink. The benefit of this approach is that the application and container no longer need to know anything about which database is present, because the database is always named "db". Consider the following:

```
declare -A env_db
env_db=([dev]=db1 [qa]=db2 [prod]=db3)
for env in dev qa prod; do
    printf "%s\n" "Deploying to ${env^^} ..."
    (cd db; ln -fs "${env_db[$env]}" db)
    export STAGE="$env" DB="db"
    ./container_deploy.sh
done
```

Likewise, you can configure your Domain Name Service (DNS) or a Virtual IP (VIP) on your network to ensure that the right database host or cluster is used for each stage. As an example, you might ensure that db.example.com resolves to a different IP address at each pipeline stage.

Sadly, the complexity of managing multiple environments never truly goes away—it

just hopefully gets abstracted to the right level for your organization. Think of your objective as similar to some object-oriented programming (OOP) best practices: you're looking to create pipelines that minimize things that change and to allow applications and tools to rely on a stable interface. When changes are unavoidable, the goal is to keep the scope of what might change as small as possible and to hide the ugly details from your tools to the greatest extent that you can.

If you have thousands or tens of thousands of servers, it's often better to change a couple DNS entries without downtime rather than rebuild or redeploy 10,000 application containers. Of course, there are always counter-examples, so consider the trade-offs and make the best decisions you can to encapsulate any unavoidable complexity.

Developing inside Your Container

I've spent a lot of time explaining how to ensure that your development containers look like the containers in use in other stages of the pipeline. But have I really described how to develop inside these containers? It turns out I've actually covered the essentials, but you need to shift your perspective a little to put it all together.

The same processes used to deploy containers in the previous sections also allow you to work inside a container. In particular, the previous examples have touched on how to bind-mount code and artifacts from the host's filesystem inside a container using the `-v` or `--volume` flags. That's how the container_deploy.sh script mounts database files on /srv/db inside the container. The same mechanism can be used to mount source code, and the Docker `exec` command then can be used to start a shell, editor or other development process inside the container.

The develop.sh utility script is designed to showcase this ability. When you run it, the script creates a Docker container and drops you into a Ruby shell inside the container. Go ahead and run ./develop.sh now:

```
#!/usr/bin/env bash

id="eed291437be80359321bf66a842d4d54"
```

```
id+="2a789e687b38c31bd1659065b2906778"
: "${IMAGE_ID:=$id}"
: "${SQLITE3_VERSION:=1.3.13}"
: "${STAGE:=dev}"
: "${DB:=db1}"

export DB STAGE

echo "Launching '$STAGE' container..."
docker run \
    -d \
    -e DB \
    -e STAGE \
    -v "${SOURCE_CODE:-$PWD}":/usr/local/src \
    -v "${DATABASE_DIR:-${PWD}/db}":/srv/db \
    --init \
    "ruby@sha256:$IMAGE_ID" \
    tail -f /dev/null >&-

container_id=$(docker ps -ql)

show_cmd () {
    enter="docker exec -it $container_id bash"
    clean="docker rm --force $container_id"
    echo -ne \
        "\nRe-enter container with:\n\t${enter}"
    echo -ne \
        "\nClean up container with:\n\t${clean}\n"
}
trap 'show_cmd' EXIT

docker exec "$container_id" \
    gem install sqlite3 -v "$SQLITE3_VERSION" >&-
```

```
docker exec \
    -e DB \
    -e STAGE \
    -it "$container_id" \
    irb -I /usr/local/src -r sqlite3
```

Once inside the container's Ruby read-evaluate-print loop (REPL), you can develop your source code as you normally would from outside the container. Any source code changes will be seen immediately from inside the container at the defined mountpoint of /usr/local/src. You then can test your code using the same runtime that will be available later in your pipeline.

Let's try a few basic things just to get a feel for how this works. Ensure that you have the sample Ruby files installed in the same directory as develop.sh. You don't actually have to know (or care) about Ruby programming for this exercise to have value. The point is to show how your containerized applications can interact with your host's development environment.

example_query.rb:

```
# Ruby module to query the table name via SQL.
module ExampleQuery
  def self.table_name
    path = "/srv/db/#{ENV['DB']}"
    db   = SQLite3::Database.new path
    sql =<<-'SQL'
      SELECT name FROM sqlite_master
       WHERE type='table'
       LIMIT 1;
    SQL
    db.get_first_value sql
  end
```

```
end
```

source_list.rb:

```
# Ruby module to list files in the source directory
# that's mounted inside your container.
module SourceList
  def self.array
    Dir['/usr/local/src/*']
  end

  def self.print
    puts self.array
  end
end
```

At the IRB prompt (`irb(main):001:0>`), try the following code to make sure everything is working as expected:

```
# returns "AppData"
load 'example_query.rb'; ExampleQuery.table_name

# prints file list to standard output; returns nil
load 'source_list.rb'; SourceList.print
```

In both cases, Ruby source code is being read from /usr/local/src, which is bound to the current working directory of the develop.sh script. While working in development, you could edit those files in any fashion you chose and then load them again into IRB. It's practically magic!

It works the other way too. From inside the container, you can use any tool or feature of the container to interact with your source directory on the host system. For example, you can download the familiar Docker whale logo and make it available to

your development environment from the container's Ruby REPL:

```
Dir.chdir '/usr/local/src'
cmd =
  "curl -sLO "             <<
  "https://www.docker.com" <<
  "/sites/default/files"   <<
  "/vertical_large.png"
system cmd
```

Both /usr/local/src and the matching host directory now contain the vertical_large.png graphic file. You've added a file to your source tree from inside the Docker container!

When you press Ctrl-D to exit the REPL, the develop.sh script informs you how to reconnect to the still-running container, as well as how to delete the container when

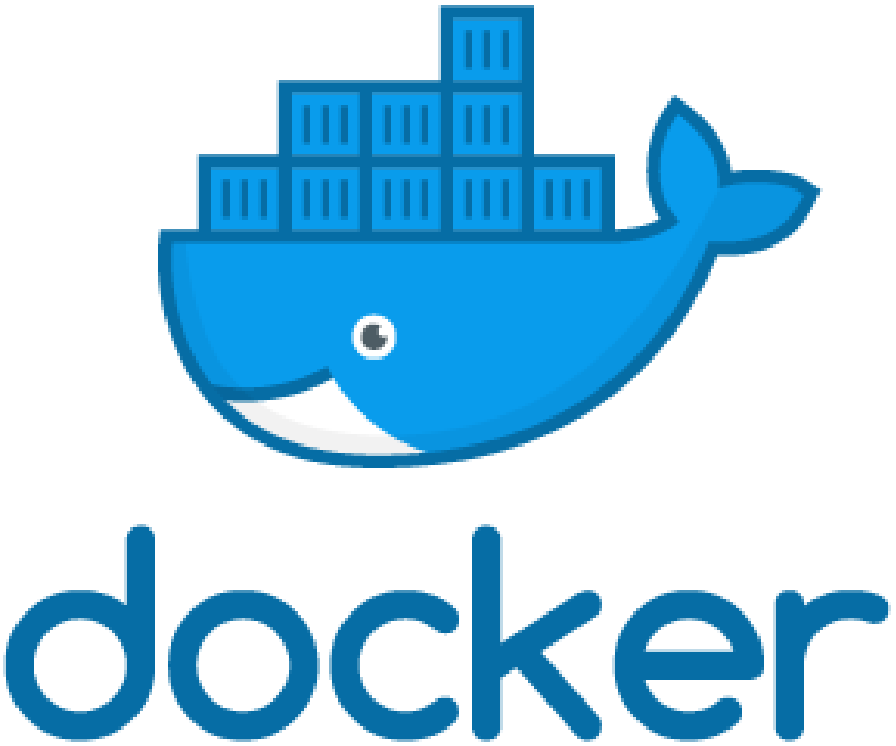


Figure 3. Docker Logo on the Host Filesystem and inside the Container

you're done with it. Output will look similar to the following:

```
Re-enter container with:
        docker exec -it 9a2c94ebdee8 bash
Clean up container with:
        docker rm --force 9a2c94ebdee8
```

As a practical matter, remember that the develop.sh script is setting Ruby's `LOAD_PATH` and requiring the sqlite3 gem for you when launching the first instance of IRB. If you exit that process, launching another instance of IRB with `docker exec` or from a Bash shell inside the container may not do what you expect. Be sure to run `irb -I /usr/local/src -r sqlite3` to re-create that first smooth experience!

Wrapping Up

I covered how Docker containers typically flow through the DevOps pipeline, from development all the way to production. I looked at some common practices for managing the differences between pipeline stages and how to use stage-specific data and artifacts in a reproducible and automated fashion. Along the way, you also may have learned a little more about Docker commands, Bash scripting and the Ruby REPL.

I hope it's been an interesting journey. I know I've enjoyed sharing it with you, and I sincerely hope I've left your DevOps and containerization toolboxes just a little bit larger in the process. ■

Todd A. Jacobs is an industry leader in DevOps transformations and the author of *The Agile CIO: Redefining IT Leadership in the Modern Enterprise*. He currently lives in Baltimore with his beautiful wife, toddler-aged son and a lovable older dog who keeps him company while he writes.

The Chromebook Grows Up

Android apps meet the desktop in the Chromebook.

By Philip Raymond

What started out as a project to provide a cheap, functional, secure and fast laptop experience has become so much more. Chromebooks in general have suffered from

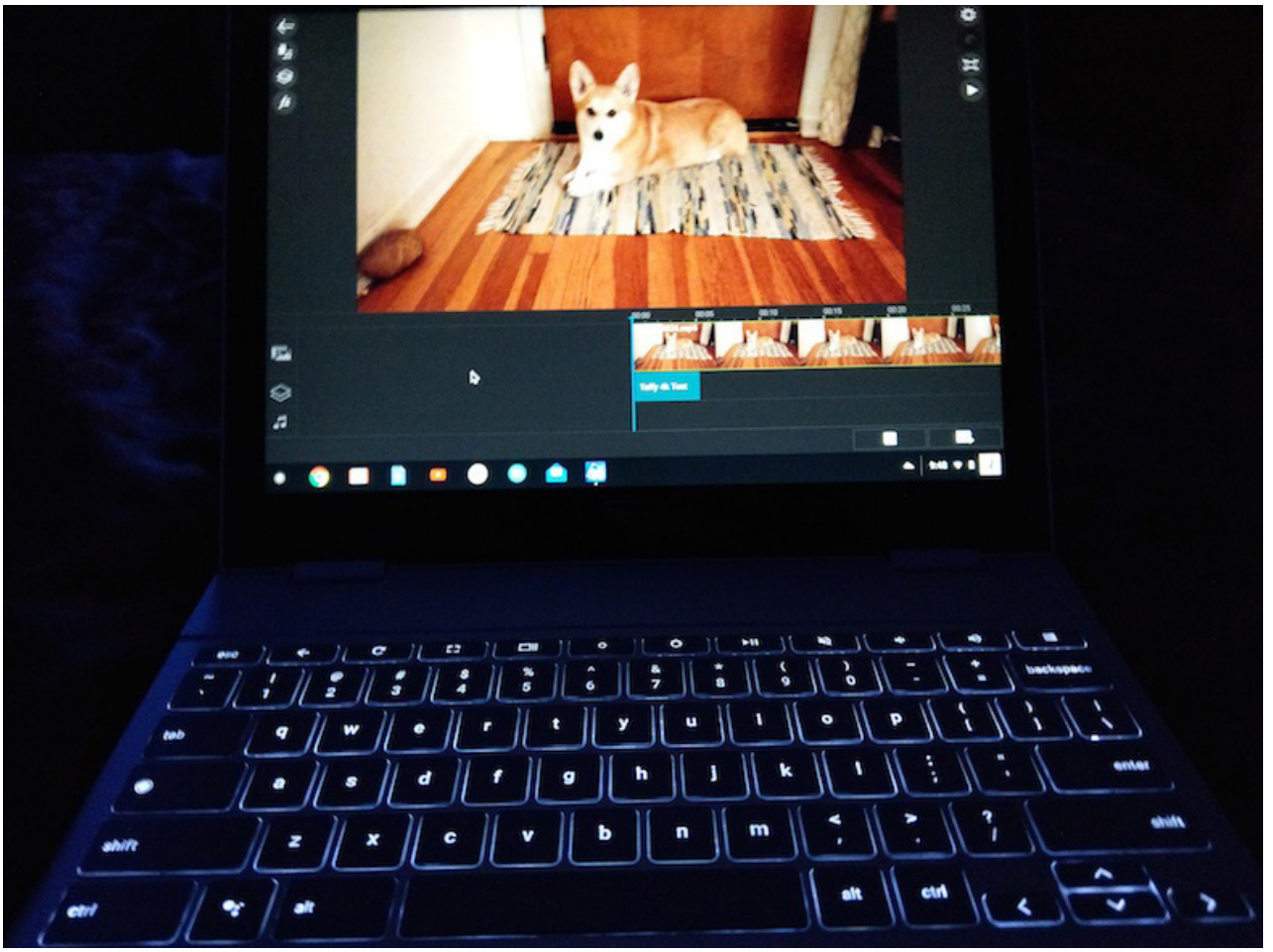


Figure 1. Pixelbook in the Dark

a lack of street-cred acceptance. Yes, they did a great job of doing the everyday basics—web browsing and…well, that was about it. Today, with the integration of Android apps, all new and recently built Chrome OS devices do much more offline—nearly as much as a conventional laptop or desktop, be it video editing, photo editing or a way to switch to a Linux desktop for developers or those who just like to do that sort of thing.

Before I go further, let me briefly describe the Linux road I've traveled, driven by my curiosity to learn and see for myself how much could be done in an Open Source world. I've used Linux and have been a Linux enthusiast ever since I first loaded SUSE in 2003. About three years later, I switched to Ubuntu, then Xubuntu, then Lubuntu, then back to Ubuntu (I actually liked Unity, even though I was fine with GNOME too). I have dual-booted Linux on several Gateway desktops and Dell laptops, with Windows on the other partition. I also have owned a Zareason laptop and most recently, a System 76 laptop—both exclusively Ubuntu, and both very sound, well-built laptops.

Then, since I was due for a new laptop, I decided to try a Chromebook, now that Android apps would greatly increase the chances of having a good experience, and I was right. Chrome OS is wicked fast, and it's never crashed in my first six months of using it. I mention this only to provide some background as to why I think Chrome OS is, in my opinion, the Linux desktop for the masses that's been predicted for as long as I've used Linux. Granted, it has a huge corporate behemoth in the form of Google behind it, but that's also why it has advanced in public acceptance as far as it has. This article's main purpose is to report on how far it has come along and what to expect in the future—it's a bright one!

Chromebooks now have access to Microsoft Office tools, which is a must for those whose employers run only MS Office products. Although Google Docs does a good job with basic document creation and conversion, and although you can create a slide presentation with it, it won't do things like watch or create a PowerPoint presentation. That's where the Microsoft PowerPoint Android app comes in handy. If you need to watch one, simply download the PowerPoint file and open it with PowerPoint (you can do this without paying for Microsoft office). However, if you

want to create or edit one, you'll have to pay for a yearly subscription or use your company's subscription.

It's also now possible to do 4k Ultra HD video editing, albeit consumer-grade video editing, not professional (think I-Movie), which addressed a reason for my own avoidance of a Chromebook in the past.

Although Android apps don't work on some older Chromebooks, there is a way to see if your older Chromebook is compatible. Go to your login picture in the lower-right corner and click on it, then go to the settings wheel in the window. In settings, if you see an option to enable the Google Play Store, your older Chromebook can run Android apps if you enable it. If you don't see this option, it can't.

What does Chrome OS still not do? High-end photo editing (aka a full Adobe Photoshop app). There is the Adobe Lightroom Android app, which is different photo-editing software that does many of the same things Photoshop does for most people. The only functions it can't do well are major retouching and graphics editing other than photos. One thing it does better than Photoshop, however, is not writing over raw images. That's the only shortcoming I can find in today's Chromebook.

Some might consider the lack of high-end gaming to be a shortcoming as well, but that's about to be addressed in the form of some very serious games coded for Android mobile that actually could be a better experience on a Chromebook (or a Chromebox for a desktop gaming experience). At the time of this writing, Epic's *Fortnite Battle Royale* is coming soon to Android, marking the beginning of what finally could bring great, well known games to Chromebooks. PUBG also is developing games for Android. Of course, keyboard and mouse/touchpad support is vital to make this truly work, and I do believe it will arrive sooner rather than later, perhaps during this year.

Some Favorite Android Apps on the Chromebook

Power Director is the first Android video-editing app I've ever used that did everything I needed, with no hiccups, and it now works on Chromebooks. Best of all,

it edits and exports to your Chromebook's internal hard drive, offline. It's also full-featured with 3D effects, titling, audio mixing and up to 4K ultra HD video exporting. Granted, to store any normal amount of video editing you'll need one that has at least 64GB, preferably 128GB or more. That means a high-end Chromebook in the range of $500–$1,000, but the point here is just that it even can be done. Getting the 4k exporting and slow-motion effects means paying for a $6 add-on pack; otherwise, Power Director is free. So, it isn't the always-free software environment to which some Linux users have grown accustomed, but $6 for Power Director's extra features is still a great value.

Now, let's talk network file management.

As is the case with all Android apps, there are many to choose from—some great,

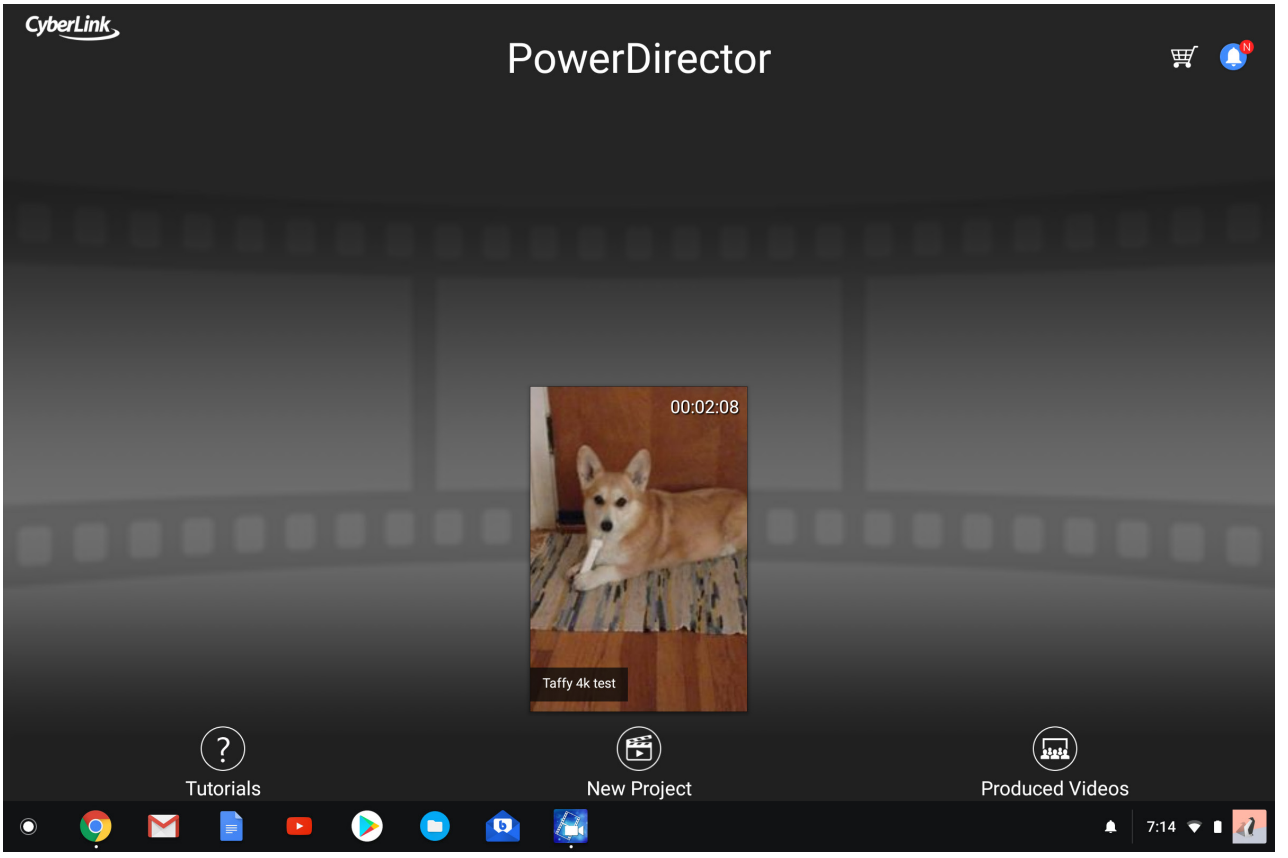


Figure 2. Power Director brings 4k Ultra HD offline video editing to Chromebooks.

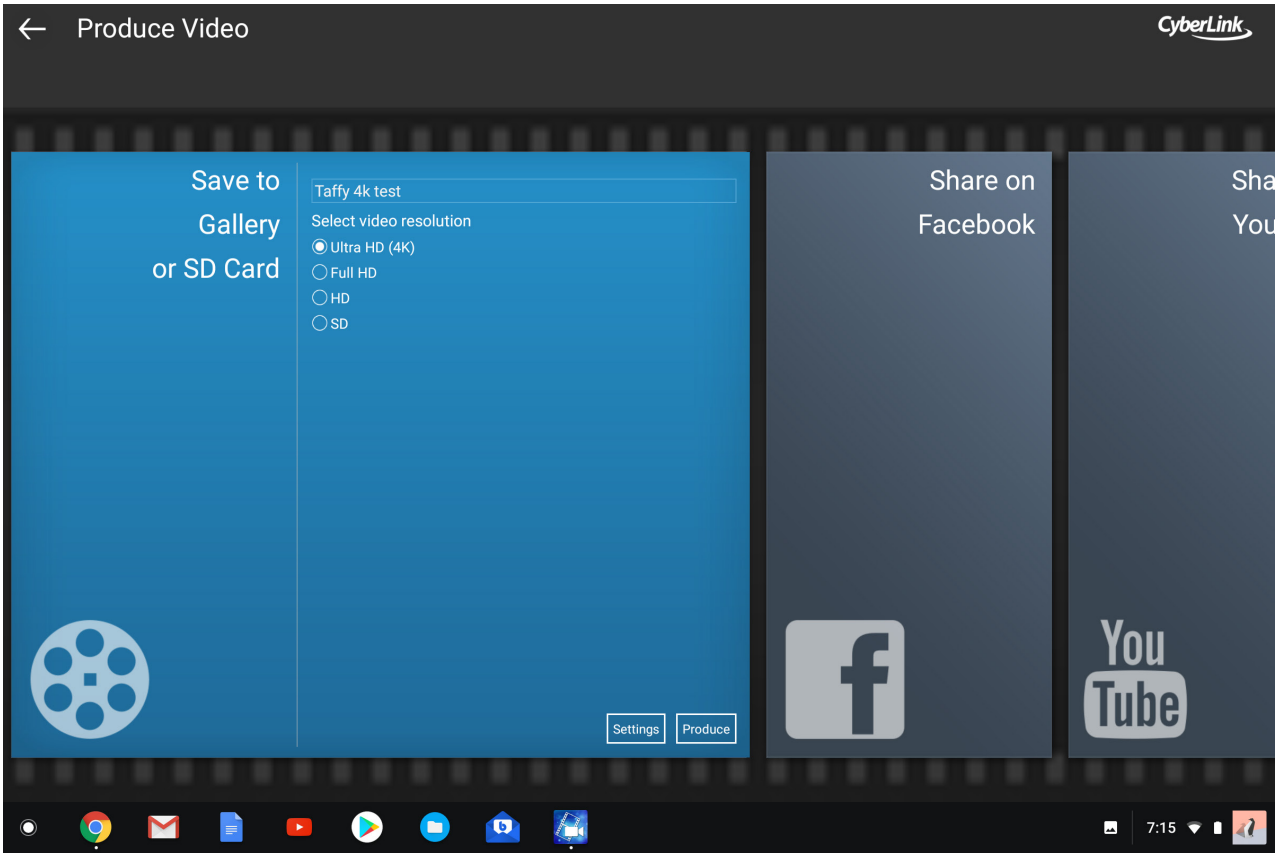


Figure 3. Export to 4k Ultra HD or lower res settings if you prefer.

some mediocre and some just plain horrible. I tried nearly a dozen different network file manager apps, and all were either terrible, didn't work or both, except one. I was very surprised that my favorite network file manager for my phone, ES File Explorer, simply would not function when installed on my Chromebook. This might be because my Chromebook is the Google Pixelbook and uses an Intel processor, and ES might work only on ARM processors, but I digress.

The one app I discovered that did everything right is called Smart File Manager. It finds all of my network drives and moves files easily back and forth from all of them. So you no longer have to depend on Google Drive to store and move your files from a Chromebook. Figure 4 is a screenshot of my network view on Smart File Manager.

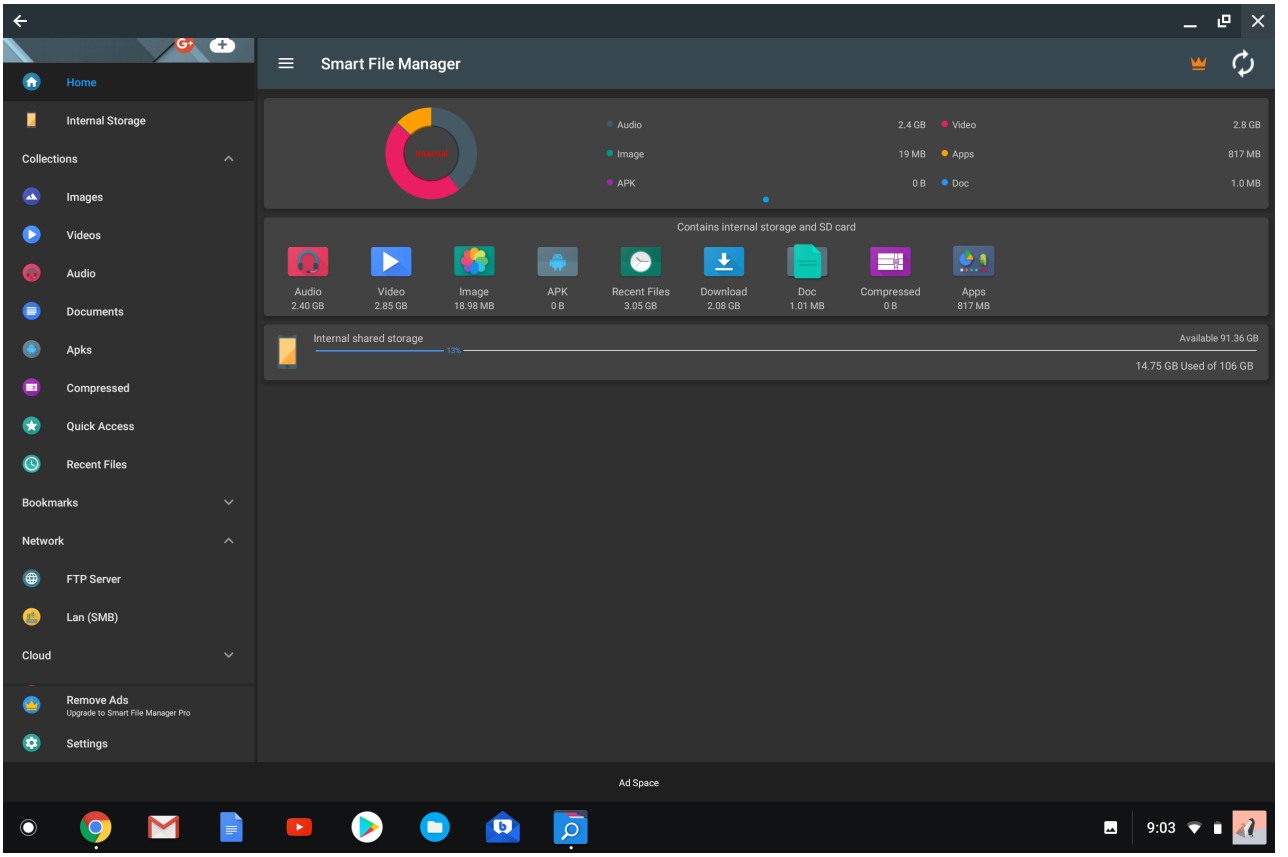


Figure 4. Smart File Manager makes it easy to move files to and from your Chromebook, via your home network.

What about email alternatives? After trying several apps, the Blue Mail app was an unexpected surprise, as I'd never heard of it before. It was the first email app I've ever installed that detected the correct incoming and outgoing IMAP email settings automatically. All I had to do was type in my email address from my ISP (not my Gmail account). If you need to access your messages offline, this is the best Android email app I could find for a Chromebook.

Personally, I like using many of the native Chrome apps that come baked in to all Chromebooks, be it the Chrome browser, Gmail, Google Docs, Google Hangouts and so on. It is nice, however, to know that many more options outside the Google ecosystem now exist should you desire to find them. However, the Google Play Store makes it easy to find Android apps, and many of them are free.

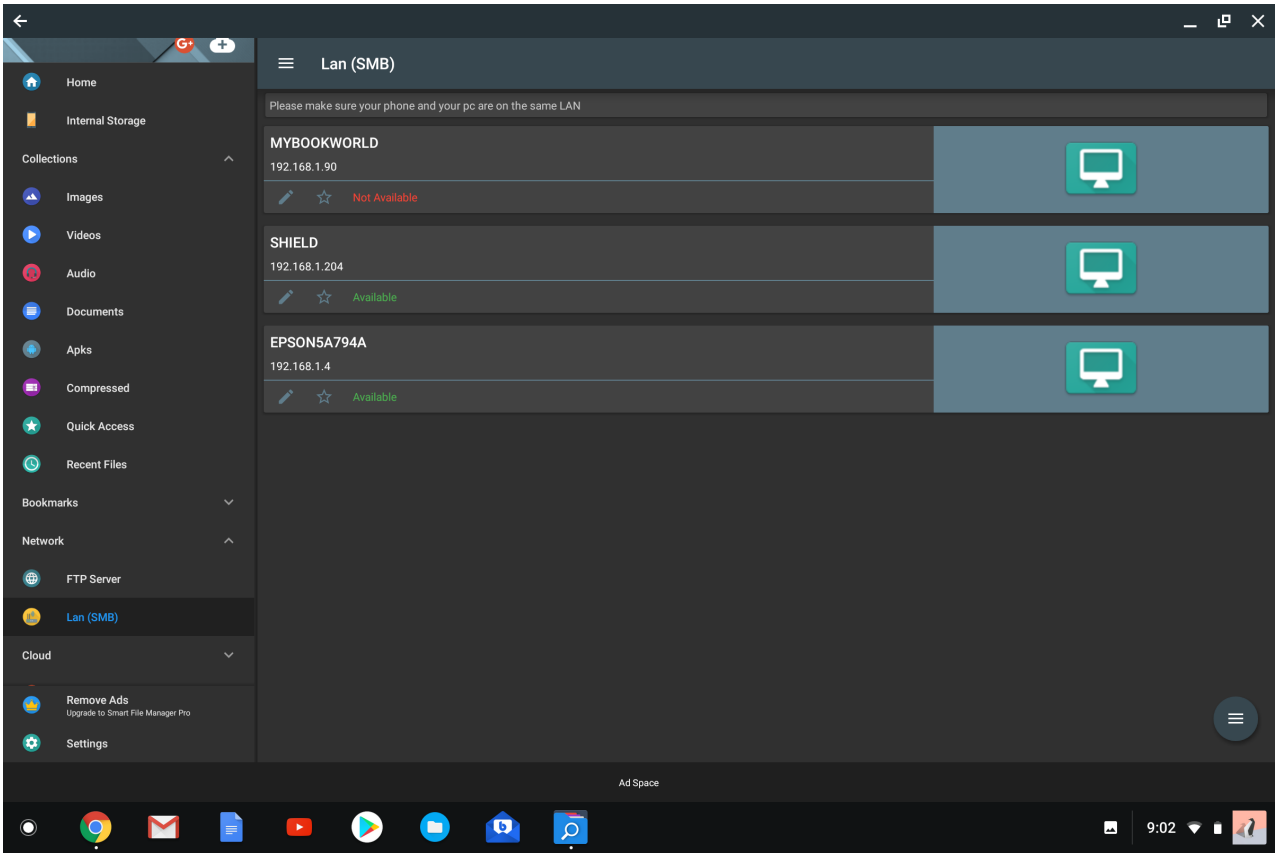


Figure 5. A user-friendly view of your network and Chromebook sources for moving files.

When you first launch an Android app, it most likely will appear in a mobile-device-size screen. When you click the resize square in the upper-right corner, it always will come with a warning that it might not function correctly in full-screen mode. I have yet to find an app I liked that had this problem, so go ahead and resize, and things probably will work as they should.

Once you open an app in full-screen mode for a Chromebook, it will default to full screen each time you open an Android app thereafter. As time goes by, this should stop being an issue, as more Android developers make screen-size detection automatic.

Tablets

Although interest in Android tablets has declined in recent years, Chrome OS is moving into that form factor with the Chrome OS experience. Acer recently introduced the first Chromebook tablet: the Acer Chromebook Tab 10. And the first detachable laptop/tablet Chromebook, the HP X2, also is now available. These are just a few examples of the inevitable broadening of the marketing scope for Chrome OS on both ends of the economic spectrum. On the other end of this spectrum is the Google Pixelbook, which was released in November 2017. High-end specs include up to an Intel I7 Kaby Lake processor with 512GB NVMe SSD of storage, 16GB of RAM and a backlit keyboard standard. I have the Intel I5, 128GB version of this, and it's joy to use everyday. For those who prefer a traditional desktop experience, there are Chromeboxes from many manufacturers in all varieties of specs and prices as well.

Distributions

If you like your Linux distro, you can keep your Linux distro. Chrome OS is more like a Linux distro than a true OS, in that it is running on the Linux operating system. This makes sense, given Chrome's roots in the open-source Chromium project. It's also why you can run Chrome OS and Xubuntu side by side in a chroot environment. Xubuntu's Xfce desktop is the default. You can run Ubuntu instead, but Unity isn't as smooth an experience as Xfce on most Chromebooks.

Crouton

If you want, you also can go without a GUI and do everything in the terminal. It does involve switching the Chromebook to developer mode, then installing Crouton from GitHub. Once done, you can hotkey between the two, with no need to dual-boot. If this interests you, you should know that some Ubuntu software hasn't been compiled to work on ARM processors, so this option is more likely to work consistently on a Chromebook with Intel x86 or AMD64 processors. And, if you wish to return to the original Chrome OS state, it's as easy as getting out of developer mode and letting Chrome OS overwrite all of the changes you made. For more information on Crouton, see the project's GitHub page.

Crostini

There is a program called Crostini that will bring a natively running Linux terminal in a container to Chromebooks. Linux apps and/or a distro can run securely in the terminal as well. Imagine all of the developer tools you need, accessed via a terminal inside the lightweight Chrome OS? No need for Crouton to get your work done as a developer. Containers in general could be the final step in getting Chromebooks on a level playing field with Windows and Mac. For example, any time you use an Android app on a Chromebook, you're running it in a container that isolates them to not affect the Chrome OS running on your Chromebook. Crostini does the same thing for developers, isolating what you're developing from the Chrome OS to maintain its stability. Crostini would make using Crouton unnecessary and would be much more secure, in that you would not need to enter into developer mode and chroot, like you do with Crouton. Once enabled, you would see the option to download Crostini in the settings menu on any non-managed Chrome OS device. Crostini is currently available in both the dev and beta channels of Chrome OS version 68 and is expected to arrive in the stable channel by version 69 in mid-September.

Chromebooks and Education

Another Chromebook foothold is devices for education. Apple recently made a publicity splash with the introduction of a $299 iPad for students to use at school and home. There was one problem with this scheme. Chromebooks—that is, functional laptops with keyboards—start at $149 and have for quite a while. A $150 price difference may not seem like much to some people, but for school districts buying many of these in bulk on the taxpayer's dime, Chromebooks are a sensible option.

Last year, Chromebooks made up 58% of computing devices shipped to schools, while Apple devices dropped 19%, down an embarrassing 50% from 2014, according to data from the market research firm Futuresource. Another fact to consider is that keyboards are a necessity for any classroom above the second grade, and the Chromebook's keyboards have stood up to that abuse for years now.

With everything being stored in the cloud, what the student does at school is easily retrieved and worked on at home as homework. Let's not forget the baked-in Google

Docs, Gmail and Classroom that students likely already are familiar with and have been since 2012. They also are all cross-platform and free, which is something Apple's walled garden of expensive software has never delivered.

My son was the first to buy a Chromebook in my family, which was his choice, but he did ask for my advice before he bought it. That was in 2013, and he still uses it every day, long after he got his bachelor's degree in 2014. He loves it and wants to buy another one soon. He's also a millennial who previously owned a MacBook, and I'm sure he's not alone in this transition of preferences. Once he discovered he could manage his iTunes playlist on a Chromebook using Spotify Premium, he was completely sold on a Chromebook! My son's experience with a Chromebook is multiplied by the thousands, maybe millions, worldwide who have had similar good experiences with a Chromebook. This, more than any other explanation, is why the Chromebook has flourished and will continue to do so well into the future.

Conclusion

So is a Chromebook right for you? As with most purchases, it has to provide the right combination of features for the price, and it also should be an improvement over what you have been using. Admittedly, it's not for everyone, but it's a good choice for a growing number of those who simply want an easy-to-use, safe and quick laptop experience. Did I mention that it boots up in five seconds to your login? And finally, for me, the introduction of Android apps was all the push I needed to buy one.

This leads to my view on the future of Chromebooks. The low-end budget-minded versions are more up to date, useful and visually attractive. However, the more enticing news is the production of higher-end, more useful and, dare I say, elegant-looking Chromebooks. Yes, they cost more, but they also do more. In the case of the newest-edition Google Pixelbook I own, they are beautiful pieces of technology anybody would be proud to bring into any boardroom of a major corporation. I know that doesn't matter to some people (myself included), but it's good to see this push in great laptop design for Chromebooks.

The focus of this article has been on what the Chromebook can now do that they

previously couldn't do. Wrap your brain around how this extremely light operating system is doing nearly as much as much more heavy, memory-intensive operating systems that tend to process slower and, in the case of Windows, need anti-virus and anti-malware software constantly running in the background to keep your experience safe, using resources that otherwise would go to whatever you're doing at the moment. This is what is called progress, and in the case of personal computing, Chromebooks are what progress looks like. The mission is far from complete, but it's already proving itself as a viable computer choice for many people.

There's also some speculation on the future of the Chrome and Android. Google is in the early stages of testing a new OS called Fuchsia that is not built on Linux. It's built on a new open-source microkernel called Zircon, and the focus is on one unified OS for mobile devices and laptops/desktops—or in Google's world, pulling Chrome OS and Android OS functions into one new OS that would replace both.

That is one gigantic undertaking, but if any company has the cash and patience to pull it off, it's Google. If it can do it, the new Fuchsiabook or Fuchsiaphone (my creation, but who knows) could be simply amazing! This is what progress looks like. ■

Philip Raymond is a Master Control Supervisor at WFLD-Fox Television in Chicago. He has used and enjoyed using Linux for 15 years and enjoys learning about new open-source projects. You can follow Phil on Twitter @tvphil or on Facebook at www.facebook.com/tvphil.


THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
AUGUST 29-31, 2018 | VANCOUVER, BC
THE PREMIER OPEN SOURCE EVENT OF THE YEAR.
Discover how to best leverage open source technologies at Open Source Summit across AI, Cloud Native, Infrastructure & Automation, Containers, Linux, DevOps and more.
2,000+ Attendees
250+ sessions, workshops, and tutorials
Featured Keynote Speakers:
Ajay Agrawal
Author, Prediction Machines: The Simple Economics of Artificial Intelligence
Austen Collins
Founder & CEO, Serverless Inc.
Linus Torvalds
Creator of Linux & Git and Fellow, The Linux Foundation
Preethi Kasireddy
Founder & Chief Executive Officer, TruStory
Van Jones
President & Founder, Dream Corps; CNN Contributor; Best-Selling Author
Window Snyder
Chief Software Security Officer, Intel
And More!
Use discount code LJ15 for 15% off
events.linuxfoundation.org/events/open-source-summit-north-america-2018/
Register Now

#geeklife: weBoost 4G-X OTR Review

Will a cellular booster help me stay connected on my epic working road trip?

By Kyle Rankin

I'm a Linux geek, and I think I safely can assume everyone reading an article in *Linux Journal* identifies themselves as Linux geeks as well. Through the years I've written about many of my geeky projects here in *Linux Journal*, such as my Linux-powered beer fermentation fridge or my 3D printer that's remotely controlled using a Raspberry Pi and Octoprint software. The thing is, my interests don't stop strictly at Linux, and I doubt yours do either. While my homebrewing, 3D printing and (more recently) RV interests sometimes involve Linux, often they don't, yet my background means I've taken a geek's perspective and approach to all of those interests. I imagine you take a similar approach to your hobbies and side projects, and readers would find some of those stories interesting, useful and inspirational.

We discussed this at *Linux Journal* and realized there should be a space for Linux geeks to tell their geeky stories even if they don't directly involve Linux. This new series, #geeklife, aims to provide a place where Linux geeks can talk about interests and projects even if they might not be strictly Linux-related. We invite you to send proposals for #geeklife articles to ljeditor@linuxjournal.com.

For this first #geeklife article, I'm telling the story of a geeky, connected working road trip I just took in my RV, and within that context, I also review a particular piece of hardware I hoped would make the trip possible, the weBoost Drive 4G-X OTR. In the interest of full disclosure, Wilson Electronics provided me with this review unit, and I did not purchase it independently.

Working Remotely

My job is 100% remote. It took me many years of braving multi-hour California Bay Area commutes and turning down opportunities to find a job where I finally could work completely from home. Smart organizations are finally beginning to realize the many advantages to having a remote workforce, but I've found it works best if you have the right team, the right tools and the bulk of the workforce is remote. When everyone is distributed, everyone relies on the incredible modern collaboration tools currently available, and you have focus and incredible productivity when you need it while still being able to communicate with your peers.

My wife is a freelance writer and has worked from her home office long before I also worked from home. Once I also landed a job where I was completely remote, we posed the following question to ourselves: in theory, we could work from anywhere with a decent internet connection, but in practice, is that really something we could do? What would that kind of working trip look like?

The key piece in the discussion of working from anywhere was our recent purchase of a used 17′ RV van. Without an RV, we would have been talking about working from various hotels, or perhaps renting a cabin for a few weeks in some remote location (but not so remote that it didn't have a solid internet connection). With the RV, a whole world of national parks, campgrounds and other destinations opened themselves up to us, so we started planning an epic two-week working road trip that would clock a few thousand miles and hit multiple states and national parks before we were done. We have a school-aged son, so even though we could work from anywhere in theory, we couldn't do it while he was still in school, so our plan was to take this trip at the start of his summer break.

It just so happened that our trip coincided with a speaking engagement I had in Utah. We live in the California Bay Area, so this helped provide a main anchor point for our trip. We would leave the Bay Area on the first weekend and drive toward Southern Utah via Las Vegas. Then we would spend a few days camping and working in Bryce Canyon. Next I would attend my conference, and then we would spend the next weekend traveling from Utah into Colorado, stay a day at Dinosaur National Quarry

in Utah and ultimately camp around Rocky Mountain National Park. From there, we would take a short drive to a friend's house, spend the night and work from there the following day, and then spend the rest of the week working outside of Aspen before making our way home the final weekend. The overall idea was to reserve our big travel days for weekends and, for the most part, keep our day-to-day travel minimal during the week, especially during our main working hours.

The Connectivity Problem

One of key challenges in making a trip like this work was the fact that both of us need a solid internet connection to do our jobs. If you were just taking a two-week vacation, you might be able to live without an internet connection the entire time, but for us, without a connection, we weren't working. Although I knew we had a few anchor points (the conference hotel and our friend's house) that would have good internet connections, the rest of the time, we would be on our own.

I had researched mobile satellite internet options in preparation for this trip, but I was stopped by the cost. Although the bandwidth in modern satellite internet services would have worked fine, you are looking at a $1,000 minimum investment to get a portable satellite dish that you have to aim yourself (which takes between 15 and 45 minutes each time). For around $6,000, you can invest in a dish that mounts to the roof of your vehicle and can aim itself automatically. In either case, you also are looking at a minimum $50-per-month metered plan with a two-year contract. While a mobile satellite dish means we literally could access the internet from anywhere in North America, we just couldn't justify the cost for a two-week trip.

Since satellite was out, that left cellular internet. We currently use Project Fi, a low-cost Google cellular service that combines the Sprint, T-Mobile and US Cellular networks into a single network and hops to whichever network happens to have the strongest connection. Project Fi allows tethering and has reasonable data rates (with a cap that converts over to "unlimited" once you reach it), so we decided just to tether from our phones for the trip. This meant we would need to make sure that if we camped at a national park, it was at a campground that had a reasonable cellular connection. When we looked at Project Fi's coverage map, it looked like most of the

places we were planning to visit had okay coverage, so we figured we would make some basic plans and in the worst case, go to an RV park instead of a traditional campground and use the RV park's (notoriously oversubscribed) WiFi.

Of course, this reliance on cellular tethering presented a dilemma. We were basing a majority of this trip, and deciding where we could say, all on whether my cell phone could get a solid connection. On top of that, my RV is a steel and fiberglass vehicle, where, when parked, it has most of the windows covered in metal Reflectix to keep the inside cool. Although the RV isn't a Faraday cage, it's not far from it. I started realizing that I needed some kind of insurance policy to make sure I got the strongest connection I could get, and that's when I started researching cellular boosters and discovered the weBoost 4G-X OTR.

The weBoost Series

It turns out that Wilson Electronics makes quite a few cellular boosters under the weBoost brand both for vehicles and homes. The thing that attracted me to the 4G-X series in particular was that although it isn't exactly cheap (retail $500), it claimed up to a 32X boost in signal, was compatible with my networks and was designed to be used in a vehicle. These boosters use an external antenna mounted on the vehicle to get the best possible signal, and that antenna connects to an amplifier that's powered off the vehicle's battery. This amplifier boosts the signal and sends it to an internal antenna you place near your phone. It's important to note that to get the benefit of the signal booster, you need to place your phone within a few feet of the internal antenna—the closer to the internal antenna the better.

The 4G-X series has a few different models in that product line. The base model is designed for car use and has a small magnetic-mount antenna designed to stick to the roof of a car. There is also an "OTR" model that is designed for truckers with a large 18″ external antenna on a pole designed to be mounted to a semi's side mirror and a small internal antenna designed to velcro next to the driver's seat. Finally, there's a newer "RV" model that removes the pole from the external antenna, so it can mount to the roof of an RV more easily and replaces the smaller internal antenna with a slightly larger one designed to sit on top of a counter, so you can aim it back at the living quarters of an RV.

I looked at each of the models, and even though I technically have an RV, I chose the OTR model (Figure 1) for a few reasons. For one, while each antenna claimed to provide similar amplification, the OTR antenna seemed the strongest. For another, I was going to have to build an antenna mount for the antenna regardless, so I thought the taller 18″ external antenna would help me to get extra height when I was parked compared to the RV antenna. Finally, I preferred the smaller internal antenna so I could use velcro to mount it behind the driving area while driving, and then re-aim the antenna if necessary while parked. Of course, if I change my mind, I always can order

Figure 1. weBoost 4G-X OTR Hardware

the other types of antennas later.

weBoost 4G-X OTR Tech Specs

Frequency:

- Band 12/17 — 700MHz
- Band 13 — 700MHz
- Band 5 — 850MHz
- Band 4 — 1700/2100MHz
- Band 2 — 1900MHz

Max Gain: 50 dB

Power Req: 6V/2A

Connectors: SMA Female

Figure 2. The OTR Antenna in "Driving Mode"

Figure 3. The OTR Antenna in "Camping Mode"

The Installation

Installing the weBoost 4G-X OTR itself was relatively straightforward and fast with the bulk of the work revolving around running cables and my desire to create a telescoping antenna mount at the back of my vehicle. Many RVs provide some kind ladder at the back of the vehicle so you can access the roof, but my van didn't, so my solution was to mount a length of PVC pipe to the spare tire bracket on my rear door, and inside it put a length of electrical conduit with a 90° curve on the end. I

Figure 4. The Amplifier Mounted on My Cabinet Wall

then could mount the antenna to the end of the conduit and while driving leave the conduit inside the PVC pipe secured with a bolt and pin that ran through both pipes (Figure 2). When I parked, I then could remove the pin, telescope the antenna an extra 4′ up and secure it back in place with the same pin (Figure 3).

Once the antenna was in place, I telescoped it to its maximum height and then ran the antenna wire through the back door, under the back seats, through the side wall and

inside a cabinet where I also mounted my TV. The amplifier is powered off a standard 12V car outlet, so I made sure to put it close enough to a 12V outlet that its (rather long) cable easily could reach. I ultimately mounted the weBoost amplifier itself just above the TV, so I can tell whether it's on and whether it's getting any interference (Figure 4).

The installation instructions state that you want to make sure the external and internal antennas are as far apart as possible (at least six feet apart) so they don't interfere with each other. If they do interfere, the amplifier will reduce its signal and an LED will blink to let you know about the problem, so when installing everything, it's important to try to mount your external and internal antennas as far away from each other as possible. In my case, I mounted the external antenna at the back of my vehicle and ran the internal antenna from a rear cabinet in the van where the amplifier was mounted toward the front of my van. That way, we not only would avoid interference, we'd also get the strongest signal boost while we were driving.

All in all, the installation took only a couple hours, the bulk of which was building my homegrown antenna mount and running wires behind walls inside the RV. Otherwise, just connecting the wires and mounting the amplifier on the wall would have taken only a few minutes. Once all the wires were run, I turned it on, and everything worked out of the box. The amplifier itself pulled about .6 amps from my battery when on, and even in my driveway, I noticed a boost inside the RV from about two bars LTE to full coverage.

The Trip

The first test of the weBoost was on our first day. We were driving from the California Bay Area through Southern California into the Mojave desert and ultimately ending the long day in Las Vegas. While I was driving the whole time, my wife was using her phone and noticed a significant improvement in her connection throughout the drive—to the point where even through some of the more remote parts of the drive, including the Mojave desert, she had a solid internet connection. This is particularly notable considering the fact that we were in a large steel vehicle that blocks quite a bit of signal.

Figure 5. A Lunchtime Stroll at Bryce Canyon

The real test was the second day when we set camp for a few days in the North campground at Bryce Canyon (Figure 5). First, I was pleasantly surprised to find that Bryce Canyon had pretty respectable cellular coverage throughout the main areas of the park—I had a bar or two of LTE in the main areas of the park near the entrance and dropped down to 3G only when I drove miles into the park. At my campsite itself, I had a few bars of LTE at -117dBm, but once I extended my antenna and turned on the booster, I placed my phone on top of the internal antenna, and that jumped up to full bars LTE at -87dBm (a 30dBm gain).

When we started working out of the campground, we quickly ran into an issue, not with our cellular network but with our WiFi. Because our computers were tethering

Figure 6. My Office at Bryce Canyon

from my cell phone, we easily could work from inside the RV, but the phone's limited WiFi range made it difficult to work too far outside. Fortunately, I also had installed a Vonets WiFi booster as part of my RV offsite backup project, so I was able to configure it to boost the cell phone's WiFi signal and set up my office outside in the shade (Figure 6). From there, I was able to work comfortably with a solid connection. In fact, on the last day at the park, I even participated in a video conference call and was told it was the clearest I had ever been.

The first time we were shocked with the weBoost performance was when we drove within Bryce Canyon to see its natural bridge. When we stepped out of the RV,

I noticed my signal had dropped to zero bars CDMA, and the connection details hopped between no signal and a very weak signal. Although I didn't particularly need the internet at that point, I couldn't help but test the booster, so I fired it up, placed my phone over the internal antenna and noticed I got a boost up to a few bars LTE.

We left Bryce Canyon to go to my conference, and once we left civilization again, we were back on the road and camped outside Dinosaur National Quarry on the border of Utah and Colorado. We stayed at an RV park, and inside our RV, we had zero signal. Outside, if we stood in the right place, we could get a bit over two bars of 3G CDMA. Once we turned on the booster and went back inside, that signal increased back to full bars LTE.

Our next stop that weekend was Rocky Mountain National Park. On our way there, I discovered the primary downside to having a cellular booster in the van: even in the middle of the Rocky Mountains my wife had plenty of bandwidth to stream John Denver. Unfortunately, once we actually got into the park itself, due to the elevation, mountains, trees and overall remoteness, most of that park has almost no cellular coverage. The first night we stayed at the West campground, and even with the booster, we were only able to wring out a weak 3G connection. The rest of the campgrounds were full, so we moved to a campground just outside the park at Estes Park to start the work week. This campground had a reasonable LTE connection at -108dBm, but we were able to boost it, again, to full bars LTE at -94dBm.

Our final remote stop was one where the weBoost made all of the difference between being able to work from a campground and not. We spent the last two working days of our trip at a KOA outside of Aspen at Crystal Spring. When we parked at our campsite, we were disappointed to find that we had a -123dBm zero bar CDMA connection, occasionally increasing to a single bar if we stood in the right place outside. We could tell when this happened, because we'd get a couple cellular notifications as we were walking around the RV and then nothing. The connection was so weak, my wife couldn't even make a phone call without it breaking up.

Clearly we couldn't work with a connection like this. Fortunately, since this was a

KOA, it did provide WiFi as a fallback that my wife tried but quickly found it slow and oversubscribed—as are most campground WiFi connections. With such a weak cellular signal, I didn't hold out much hope for the weBoost but decided to try it anyway. With the booster on and my phone on top of the internal antenna, that -123dBm weak CDMA connection turned into a -90dBm full bars LTE connection, and we were able to work out of the RV and place clear phone calls just like at home.

The Verdict

Now, although this cellular booster claims to boost your signal, it doesn't claim to be magic. It can amplify a cellular signal, but the signal has to be there to begin with. Even if it could provide its maximum stated boost of 32x, 32 times zero is still zero. This matched my own personal experiences. Throughout my trip, I ended up in areas with absolutely no civilization and no cellular signal, such as the long drive through the northern Nevada prairie. The weBoost couldn't help me there. Yet in every case where I had some sliver of a signal, the weBoost managed to turn that into a dramatically improved network—in some cases maxing it out!

Both my wife and I came away very impressed with the weBoost 4G-X OTR and felt like it made a working road trip like this possible. There were many instances where we would have been on an incredibly weak internet connection or otherwise would have had to move to some other campground without the weBoost. Another less obvious quality-of-life improvement was the fact that we ended up with a much stronger cellular connection while we were driving as well. You don't realize how often you rely on an internet connection on the road until you are on a long road trip and need to look up campgrounds while you are driving to your destination.

The real question is whether it's worth the $500 price tag. I suppose that decision comes down to your budget and how frequently you take trips where having a strong cellular connection is critical. If you are someone who works from the road most of the time or takes a lot of road trips, the investment is definitely worth it. If you are pricing out satellite hardware and are suffering from sticker shock, a cellular booster like this could help bridge the gap. Although it won't bring you the internet out in the middle of nowhere like with a satellite, the hardware is much cheaper, connects

instantly and you can use your existing cellular plan.

If you are someone like us who intends to take a few long working trips and vacations like this throughout the year, I think it's worth the investment. There's nothing more frustrating than being on the road, needing a strong signal and not having it, or worse, having a strong signal and losing it in the middle of looking something up or being on a call. It's also nice having the option to camp and work further out in nature while still having a strong signal. It's easy to set up, works as advertised, and it made a definite difference in our trip, so I can safely say I'd recommend it to others. ■

Kyle Rankin is a Tech Editor and columnist at *Linux Journal* and the Chief Security Officer at Purism. He is the author of *Linux Hardening in Hostile Networks*, *DevOps Troubleshooting*, *The Official Ubuntu Server Book*, *Knoppix Hacks*, *Knoppix Pocket Reference*, *Linux Multimedia Hacks* and *Ubuntu Hacks*, and also a contributor to a number of other O'Reilly books. Rankin speaks frequently on security and open-source software including at BsidesLV, O'Reilly Security Conference, OSCON, SCALE, CactusCon, Linux World Expo and Penguicon. You can follow him at @kylerankin.

Resources

weBoost 4G-X OTR Product Page

"DIY RV Offsite Backup and Media Server" by Kyle Rankin (*LJ*, June 2018)

"A Look at Google's Project Fi" by Shawn Powers (*LJ*, July 2018)

We'd Like to Hear Your #geeklife Stories!

Send article proposals to ljeditor@linuxjournal.com.

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

What Does "Ethical" AI Mean for Open Source?

Artificial intelligence is a threat—and an opportunity—for open source.

By Glyn Moody

Glyn Moody has been writing about the internet since 1994, and about free software since 1995. In 1997, he wrote the first mainstream feature about GNU/Linux and free software, which appeared in *Wired*. In 2001, his book *Rebel Code: Linux And The Open Source Revolution* was published. Since then, he has written widely about free software and digital rights. He has a blog, and he is active on social media: @glynmoody on Twitter or identi.ca, and +glynmoody on Google+.

It would be an understatement to say that artificial intelligence (AI) is much in the news these days. It's widely viewed as likely to usher in the next big step-change in computing, but a recent interesting development in the field has particular implications for open source. It concerns the rise of "ethical" AI.

In October 2016, the White House Office of Science and Technology Policy, the European Parliament's Committee on Legal Affairs and, in the UK, the House of Commons' Science and Technology Committee, all released reports on how to prepare for the future of AI, with ethical issues being an important component of those reports. At the beginning of last year, the Asilomar AI Principles were published, followed by the Montreal Declaration for a Responsible Development of Artificial Intelligence, announced in November 2017.

Abstract discussions of what ethical AI might or should mean

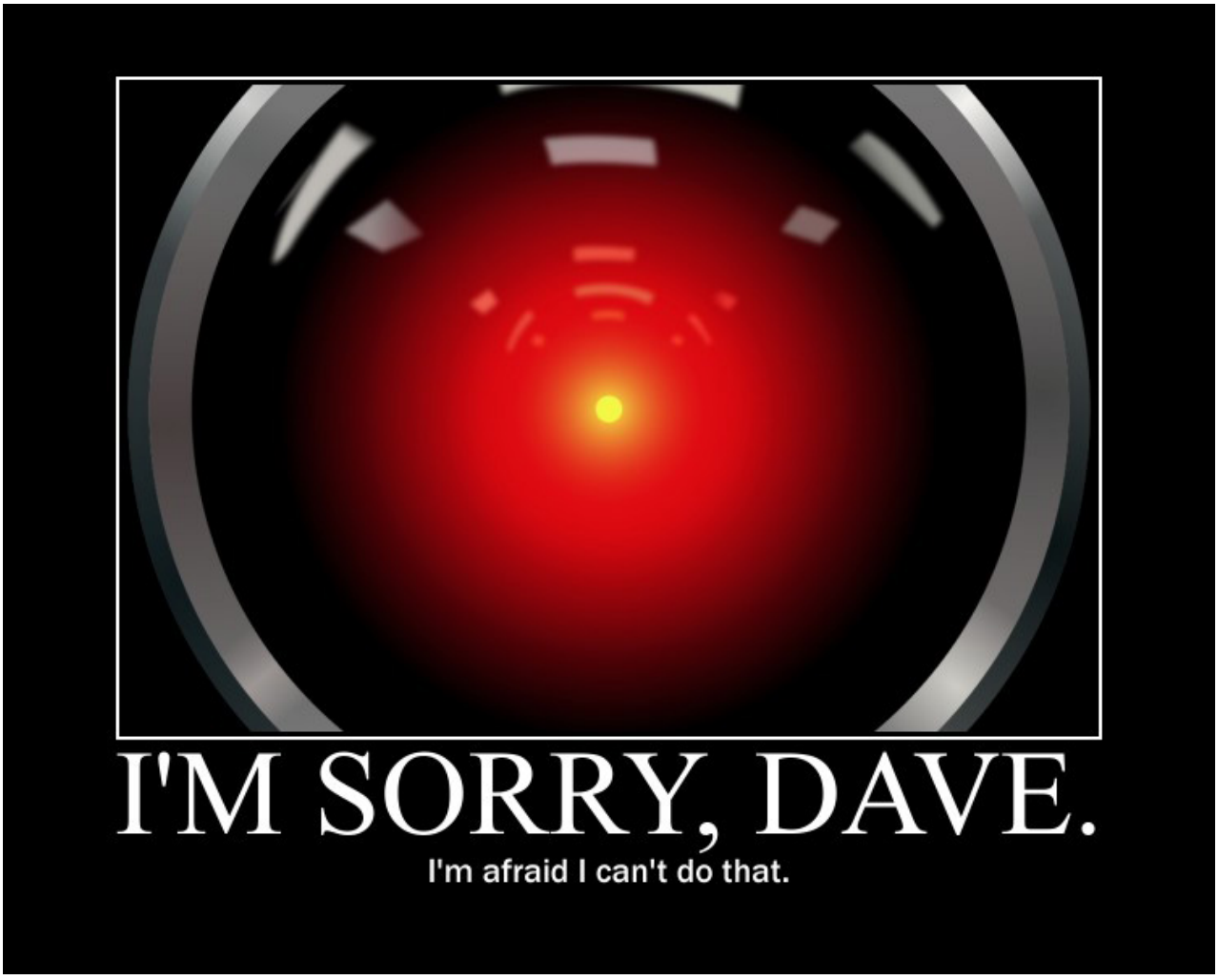


Image attribution: Cryteria

became very real in March 2018. It was revealed then that Google had won a share of the contract for the Pentagon's Project Maven, which uses artificial intelligence to interpret huge quantities of video images collected by aerial drones in order to improve the targeting of subsequent drone strikes. When this became known, it caused a firestorm at Google. Thousands of people there signed an internal petition addressed to the company's CEO, Sundar Pichai, asking him to cancel the project. Hundreds of researchers and academics sent an open letter supporting them, and some Google employees resigned in protest.

It later emerged that Google had hoped to win further defense work worth

hundreds of millions of dollars. However, in the face of the massive protests, Google management announced that it would not be seeking any further Project Maven contracts after the present one expires in 2019. And in an attempt to answer criticisms that it was straying far from its original "don't be evil" motto, Pichai posted "AI at Google: our principles", although some were unimpressed.

Amazon and Microsoft also are grappling with similar issues about what constitutes ethical use of their AI technologies. But the situation with Google is different, because key to the Project Maven deal with the Pentagon is open-source software—Google's TensorFlow:

> ...an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs [tensor processing units]), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning, and the flexible numerical computation core is used across many other scientific domains.

It's long been accepted that the creators of open-source projects cannot stop their code from being used for purposes with which they may not agree or even strongly condemn—that's why it's called free software. But the use by Google of open-source AI tools for work with the Pentagon does raise a new question. What exactly does the rise of "ethical" AI imply for the Open Source world, and how should the community respond?

Ethical AI represents a significant opportunity for open source. One important aspect of "ethical" is transparency. For example, the Asilomar AI Principles include the following:

> 7) **Failure Transparency**: If an AI system causes harm, it should be possible to ascertain why.

8) **Judicial Transparency**: Any involvement by an autonomous system in judicial decision-making should provide a satisfactory explanation auditable by a competent human authority.

More generally, people are recognizing that "black box" AI approaches are unacceptable. If such systems are to be deployed in domains where the consequences can be serious and dangerous—perhaps a matter of life or death, as in drone attacks—independent experts must have the ability to scrutinize the underlying software and its operation. The French and British governments already have committed to opening up their algorithms in this way. Open-source software provides a natural foundation for an ethical approach based on transparency.

The current interest in ethical AI means the Open Source community should push for the underlying code to be released under a free software license. Although that goes beyond simple transparency, the manifest success of the open-source methodology in every computing domain (with the possible exception of the desktop), lends weight to the argument that doing so is good not just for transparency, but for efficiency too.

However, as well as a huge opportunity, AI also represents a real threat to free software—not directly, but by virtue of the fact that most of the big breakthroughs in the field are being made by companies with extensive resources. They naturally are interested in making money from AI, so they see it ultimately as just part of the research and development work that will lead to new products. That contrasts with Linux, say, which is first and foremost a community project that involves large-scale—and welcome—collaboration with industry. Currently missing are major open-source AI projects running independently of any company.

There have been some moves to bring the worlds of open source and AI together. For example, in March 2018, The Linux Foundation launched the LF Deep Learning Foundation:

> ...an umbrella organization that will support and sustain open source innovation

> in artificial intelligence, machine learning, and deep learning while striving to make these critical new technologies available to developers and data scientists everywhere.
>
> Founding members of LF Deep Learning include Amdocs, AT&T, B.Yond, Baidu, Huawei, Nokia, Tech Mahindra, Tencent, Univa, and ZTE. With LF Deep Learning, members are working to create a neutral space where makers and sustainers of tools and infrastructure can interact and harmonize their efforts and accelerate the broad adoption of deep learning technologies.

As part of that initiative, The Linux Foundation also announced Acumos AI:

> …a platform and open source framework that makes it easy to build, share, and deploy AI apps. Acumos standardizes the infrastructure stack and components required to run an out-of-the-box general AI environment. This frees data scientists and model trainers to focus on their core competencies and accelerates innovation.

Both of those are welcome steps, but the list of founding members emphasizes once more how the organization is dominated by companies—many of them from China, which is emerging as a leader in this space. That's no coincidence. As the "Deciphering China's AI Dream" report explains, the Chinese government has made it clear that it wants to be an AI "superpower" and is prepared to expend money and energy to that end. Things are made easier by the country's limited privacy protection laws. As a result, huge quantities of data, including personal data, are available for training AI systems—a real boon for local researchers. Crucially, AI is seen as a tool for social control. Applications include pre-emptive censorship, predictive policing and the introduction of a "social credit system" that will constantly monitor and evaluate the activities of Chinese citizens, rank their level of trustworthiness and reward or punish them accordingly.

Given the Chinese authorities' published priorities, it is unlikely that the development of AI technologies by local companies will pay more than lip service

to ethical issues. As the recent incidents involving Google, Amazon and Microsoft indicate, it's not clear that Western companies will do much better. That leaves a vitally important role for open source—to act as beacon of responsible AI software development. That can be achieved only if leaders step forward to propose and initiate ambitious AI projects, and if the coding community embraces and helps realize those plans. If this doesn't happen, 30 years of work in freeing software and its users could be rendered moot by a new generation of inscrutable black boxes running closed-source code—and the world. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.