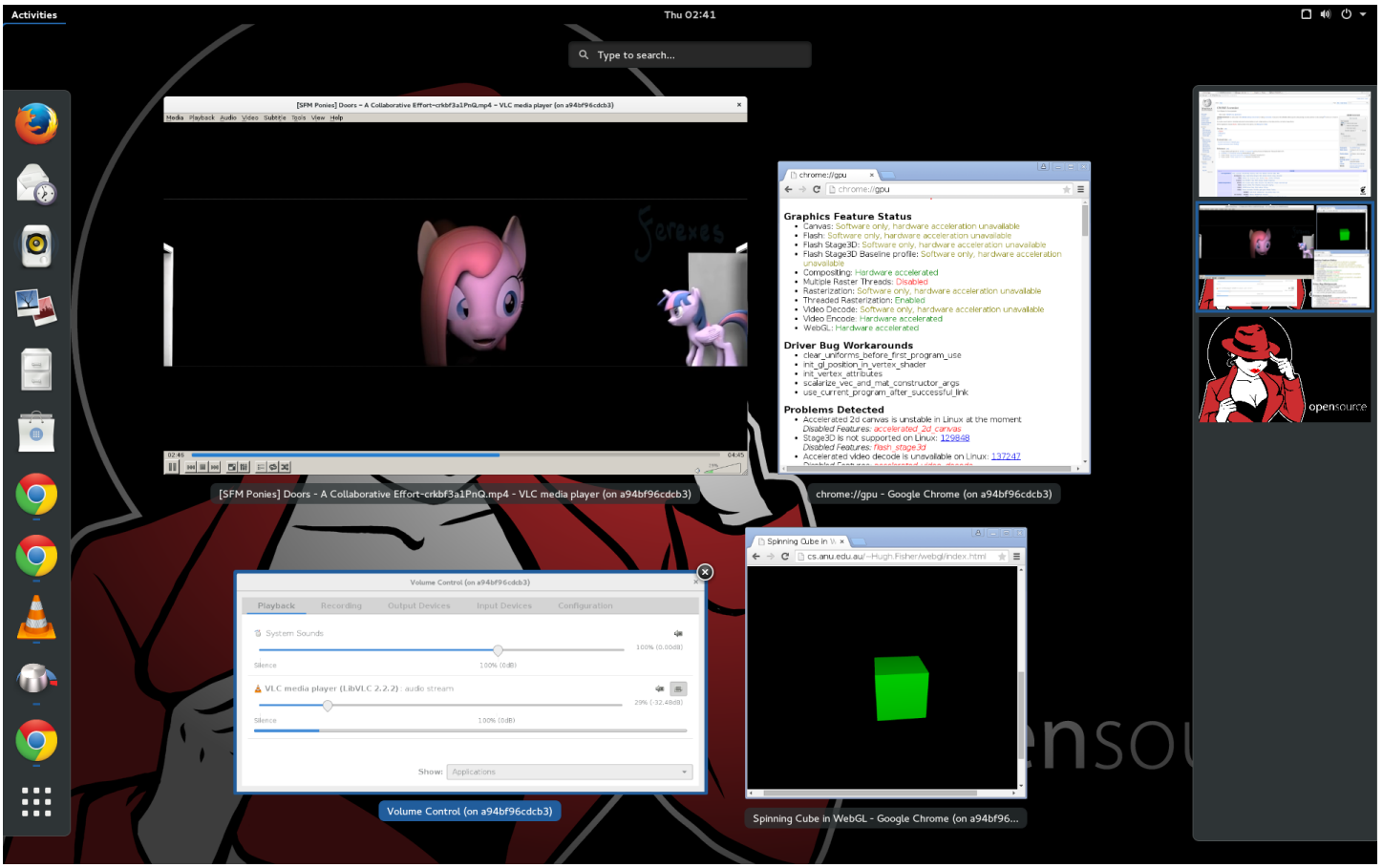

Here a completely contained GNOME runs VLC, and two Chrome containers. All have working hardware acceleration. Notice that `pavucontrol` deals with the sound output of VLC as if it were native despite that `pavucontrol` and VLC are in separate containers. Similarly, notice that GNOME composition works throughout.

The Atomic Desktop

With it still being early days for containers as a deployment technology, by far the most common use-case for containers we at Red Hat have seen is distributing "canned" runtime environments. This can speed up the on-boarding process for new or re-allocated coders, sometimes putting developers weeks ahead of competitors. It's a powerful workflow, but one that holds containers at arms reach behind `vagrant up`s, virtualization, or possibly even other operating systems. For those of us who prefer greater immersion, the question is: can this idea be take further? Can whole development environments---nay---the entire "at seat" experience be contained? With just a bit of work, the answer is a resounding yes. In fact, the "Atomic Desktop" enables exciting new workflows and control over the desktop experience previously impossible.

GNOME and First Steps

When I first considered taking the #containerchallenge, my end goal was to replicate and eventually improve on my current GLSL development environment:

- A hardware accelerated desktop and shell

- hardware accelerated web browsing with sound in both FireFox and Chrome
- DRMed Internet Radio (Q104.3 is essential to real productivity)
- Access to Windows through virtualization for further compatibility testing.

Ideally, I'd be able to run different browsers in different containers but retain hardware acceleration and hardware sound mixing. This last point is very important to me because I rely heavily on the system bell and being able to hear my iPhone's alerts as part of my workstation's sound mix.

Getting Started

Before I walk through how I was able to meet and exceed my requirements with just the standard Atomic install, I should mention that I will be covering the ideal case. Just as they do not contain, containers also do not abstract. When you ask that contained applications interact with specific hardware (such as a particular GPU), you are necessarily making your container less generic. There are ways to mitigate this, which I will cover. For the moment though, I'll consider the easy case of on-board Intel acceleration and a generically pulseaudio supported soundcard. Ease of a contained "seat" experience is yet another reason it's important to use open-source friendly hardware.

There is also much room for automation within the procedure's I'm currently using. Part of why I have shied away from---for example---using the atomic command is that I will often customize the commands used here depending on what I need at any particular moment. Such is one of the flexible advantages of a contained desktop.

Throughout my example Dockerfiles, I'll add and use my `tjay` user account. You can simply substitute your own username.

Preparing Atomic

To start simply:

- Installed Atomic on metal via the Anaconda installer
- `sudo rpm-ostree upgrade`
- `sudo systemctl reboot`

as usual.

Retrieving a Base Image

1. `curl -o 'Fedora-Docker-Base-22-20150521.x86_64.tar.xz' 'http://download.fedoraproject.org/pub/fedora/linux/releases/22/Docker/x86_64/Fedora-Docker-Base-22-20150521.x86_64.tar.xz'`

2. `md5sum Fedora-Docker-Base-22-20150521.x86_64.tar.xz`
3. `xz -d Fedora-Docker-Base-22-20150521.x86_64.tar.xz`
4. `sudo docker load < Fedora-Docker-Base-22-20150521.x86_64.tar`

Of course, compare the hash in step 2 to a published hash whose signature you have verified.

Preparing an RPM-cache

For reasons that will become apparent, the key to a smooth Atomic Desktop is a local RPM-cache. I begin by building the following Dockerfile, which I will tag as a "squid" image:

```
from Fedora-Docker-Base-22-20150521.x86_64
RUN dnf -y update && dnf -y clean all
RUN dnf -y install squid && dnf -y clean all
RUN adduser tjay ; usermod -a -G squid tjay
RUN chmod g+rw /var/spool/squid/ ; chmod g+rw -R /var/log/squid
RUN cat /etc/squid/squid.conf | sed 's/#cache_dir.*/cache_dir ufs \/var\/spool\/squid 256
00 16 256/g' > /etc/squid/tmp.conf ; echo "maximum_object_size 1024 MB" >> /etc/squid/tm
p.conf ; echo "pid_filename /home/tjay/squid.pid" >> /etc/squid/tmp.conf ; mv -f /etc/squ
id/tmp.conf /etc/squid/squid.conf
```

You will need a persistent directory (I use `/home/tjay/shared/configs/squid`) to exist on the host (ideally as the same UID/GID) and be labeled as `svirt_sandbox_file_t`. Once it does, switch to a new virtual terminal and run the squid image:

- `sudo docker run -i -t -v /home/tjay/shared/configs/squid:/var/spool/squid -u tjay squid squid -NCd1`

Preparing the GNOME and Chrome images

- use `sudo docker ps | grep -i squid` to find the name of the running `squid` (for example `drunk_sammet`).
- use `sudo docker inspect drunk_sammet | grep IPAddress` to find the IP address.
- substitute that IP for 172.17.0.1 in the following Dockerfiles:

Build and tag this first file as `gnome`.

```
from Fedora-Docker-Base-22-20150521.x86_64
```

```
RUN echo "proxy=http://172.17.0.1:3128" >> /etc/dnf/dnf.conf
RUN for I in /etc/yum.repos.d/*.repo; do cat $I | sed 's/metalink/#metalink/g' | sed
's/#baseurl/baseurl/g' | sed 's/download.fedoraproject.org\/pub\/fedora\/linux/ftp.iine
t.net.au\/linux\/redhat-fedora/g' > /etc/yum.repos.d/tmp.repo; mv -f /etc/yum.repos.d/tm
p.repo $I; done
RUN dnf -y update && dnf -y clean all
RUN dnf -y install Xorg glx-utils gnome-shell pulseaudio passwd sudo screen alsa-utils gu
ake @gnome gnome-tweak-tool xbacklight docker-io atomic pulseaudio-utils xcalib socat pav
ucontrol xscreensaver libXxf86vm libXrandr && dnf -y clean all
RUN (cd /lib/systemd/system/sysinit.target.wants/; for i in *; do [ $i == systemd-tmpfile
s-setup.service ] || rm -f $i; done); rm -f /lib/systemd/system/multi-user.target.want
s/*;rm -f /etc/systemd/system/*.wants/*;rm -f /lib/systemd/system/local-fs.target.want
s/*; rm -f /lib/systemd/system/sockets.target.wants/*udev*; rm -f /lib/systemd/system/soc
kets.target.wants/*initctl*; rm -f /lib/systemd/system/basic.target.wants/*;rm -f /lib/sy
stemd/system/anaconda.target.wants/*;
RUN rm -rf /etc/systemd/system/systemd-remount-fs.service; rm -rf /etc/systemd/system/sys
temd-journald.socket; rm -rf /etc/systemd/system/systemd-journald.service; rm -rf /etc/sy
stemd/system/systemd-journald-dev-log.socket; rm -rf /etc/systemd/system/systemd-journal
d-audit.socket; rm -rf /etc/systemd/system/systemd-journal-flush.service; ln -s /dev/null
/etc/systemd/system/systemd-remount-fs.service; ln -s /dev/null /etc/systemd/system/syste
md-journald.socket; ln -s /dev/null /etc/systemd/system/systemd-journald.service; ln -s
/dev/null /etc/systemd/system/systemd-journald-dev-log.socket; ln -s /dev/null /etc/syste
md/system/systemd-journald-audit.socket; ln -s /dev/null /etc/systemd/system/systemd-jour
nal-flush.service; rm -rf /etc/systemd/system/upower.service; rm -rf /etc/systemd/syste
m/systemd-logind.service; ln -s /usr/lib/systemd/system/upower.service /etc/systemd/syste
m/upower.service; ln -s /usr/lib/systemd/system/systemd-logind.service /etc/systemd/syste
m/systemd-logind.service;
#ADD xflux /usr/bin/xflux
#RUN echo -e '#!/bin/bash\nxflux -l '"'"'-27.46953'"'"' -g '"'"'153.02782'"'"'' > /usr/bi
n/xflux.sh
#RUN chmod a+x /usr/bin/xflux*
RUN dnf -y remove PackageKit-command-not-found && dnf -y clean all
RUN mkdir -p /run/udev; mkdir -p /run/dbus; mkdir -p /run/systemd/system
RUN cp /usr/share/zoneinfo/Australia/Brisbane /etc/localtime
RUN adduser tjay ; usermod -a -G video tjay ; usermod -a -G audio tjay
RUN sed -e 's/^root.*/root\tALL=(ALL)\tALL\ntjay\tALL=(ALL)\tALL/g' /etc/sudoers > /etc/s
udoers.new ; mv /etc/sudoers.new /etc/sudoers
RUN cat /etc/bashrc | sed 's/\(.*PROMPT_COMMAND=\).*033k.*/\1'"'"'printf "\\03
```

```
3]0;%s@%s:%s\\033\\\\" "${USER}" "${HOSTNAME%%.*}" "${PWD\/#$HOME\/~}"'"'"'/g' > /etc/tmp
; mv /etc/tmp /etc/bashrc
#RUN dnf -y install xorg-x11-drivers mesa-dri-drivers && dnf -y clean all
```

This second file, build and tag as `chrome`. You will need a Chrome RPM in the build directory for the `ADD` command to work.

```
from Fedora-Docker-Base-22-20150521.x86_64
RUN echo "proxy=http://172.17.0.1:3128" >> /etc/dnf/dnf.conf
RUN for I in /etc/yum.repos.d/*.repo; do cat $I | sed 's/metalink/#metalink/g' | sed
's/#baseurl/baseurl/g' | sed 's/download.fedoraproject.org\/pub\/fedora\/linux/ftp.iine
t.net.au\/linux\/redhat-fedora/g' > /etc/yum.repos.d/tmp.repo; mv -f /etc/yum.repos.d/tm
p.repo $I; done
RUN dnf -y update && dnf -y clean all
RUN dnf -y install  pulseaudio pavucontrol openvpn tar && dnf clean all
RUN cp /usr/share/zoneinfo/Australia/Brisbane /etc/localtime
RUN adduser tjay ; usermod -a -G video tjay ; usermod -a -G audio tjay
RUN curl -o '/home/tjay/dotjs-1.0.2.tar.gz' 'https://pypi.python.org/packages/source/d/do
tjs/dotjs-1.0.2.tar.gz'                          RUN tar xzf /home/tjay/dotjs-1.0.2.tar.gz
-C /home/tjay
RUN chown tjay:tjay -R /home/tjay/dotjs-1.0.2
ADD google-chrome-stable_current_x86_64.rpm /home/tjay/google-chrome-stable_current_x86_6
4.rpm
RUN dnf -y install /home/tjay/google-chrome-stable_current_x86_64.rpm && dnf clean all
#RUN dnf -y install xorg-x11-drivers mesa-dri-drivers && dnf -y clean all
```

Notes on the Build Files

There are a couple of tricks to note here. The first is that---as mentioned---they add a proxy setting to dnf. As part of that, the first part of the Dockerfile forces a particular Fedora mirror in each `.repo` file. This ensures repeat pulls will have the same URL. As it's highly doubtful the iiNet mirror is your most performant, you'll want to customize this portion of the Dockerfile along with the username and timezone information.

The `gnome` Dockerfile is more interesting. It concentrates on stripping down systemd to the bare essentials (including disabling journald) and ensuring that all of the directory structures systemd expects are present (at least, the ones we won't be injecting at runtime). Key to ensuring hardware acceleration and mixing is possible is that our container user is a member of both the audio and video groups, and that these GIDs match the Atomic host (which they will if you use Fedora and Fedora Atomic). Notice

that both files have a commented dnf command that would install x11 packages. More on this later.

Create a Shared `/tmp`

To facilitate X11 composition, we need a Docker hosted `/tmp` directory that our containers can share. You might think that we could just share the host `/tmp`, but that won't work. The reason is that just as root on a Fedora machine isn't *really* root (because of SELinux considerations) `--privileged` on Atomic is not *completely* privileged. The relevant restriction here is that containers aren't allowed to arbitrarily open socket files. We can bypass this restriction by hosting `/tmp` in a Docker volume instead of on the host.

To create the volume, we simply start and log out of an appropriately named container:

- `sudo docker run -i -t -v /tmp --name common_tmp Fedora-Docker-Base-22-20150521.x86_64 /bin/bash`

Starting a GNOME Environment

Enough prepping, let's get ready to start GNOME!

First, we'll need to start an interactive session in a virtual terminal. Some of the prep can be automated but the session has to be interactive so that it can take proper control of TTYs and other resources. Speaking of which, a word of warning: having a container subsume control of the "real" machine (i.e. the keyboard, mouse) means that we have to make sure to properly close out the container, less we create a situation where physical login is impossible.

Again, the GNOME container is hardware dependent, but a common run command is:

- `sudo docker run -i -t --privileged -v /dev/dri:/dev/dri -v /dev/snd:/dev/snd -v /dev/shm:/dev/shm -v /var/run/udev:/run/udev -v /var/run/docker:/run/docker -v /var/run/docker.sock:/run/docker.sock -v /dev/input:/dev/input -v /sys/fs/cgroup:/sys/fs/cgroup --volumes-from common_tmp --link drunk_sammet:rpmcache gnome /bin/bash`

where `drunk_sammet` is the name of the running squid.

If you just built `gnome` the proxy IP is correct. Otherwise, use vi to edit `/etc/dnf/dnf.conf`'s proxy line to read: `proxy=http://rpmcache:3128`.

Again, for the simple case of an Intel GPU, install the appropriate drivers now:

- `dnf -y install xorg-x11-drivers mesa-dri-drivers`

Now you're ready to start systemd:

- `/usr/lib/systemd/systemd --system &`

And to widen the `system_bus_socket` permissions:

- `chmod a+w /var/run/dbus/system_bus_socket`

Now you can start X:

- `X &`

This will almost surely take control away from your current virtual console, so you'll need to cycle through to find it again (CTRL+ALT+F1, CTRL+ALT+F2, etc.).

Once you're back at the console that you used to start X, you're ready to setup and become your preprepared user (in my case `tjay`). Don't forget to set a password for them.

- `passwd tjay`
- `su tjay -`

Now the moment of truth:

- `export DISPLAY=":0"`
- `gnome-session`

If everything went correctly, you can now find the X virtual console (again, CTRL+ALT+F1, CTRL+ALT+F2, etc.) and watch as GNOME starts up. From now on, these instructions are meant to be followed from *within* GNOME.

Starting Chrome

Once you've gone through the GNOME starting slides (setting your language, etc.). You're ready to try launching a hardware accelerated container. Launch and pull-up the Guake console (the default key for Guake is F12).

First let's make a note of the current `gnome` container's `/etc/machine-id` and pulse entry:

- `sudo cat /etc/machine-id`
- `sudo ls /tmp/pulse-*`

Then we're ready to launch Chrome. Again, assuming simple Intel hardware:

- `docker run -i -t --privileged -v /dev/dri:/dev/dri -v /dev/snd:/dev/snd -v /dev/shm:/dev/shm --ipc=container:ecstatic_sinoussi --link drunk_sammet:rpmcache --volumes-from common_tmp chrome /bin/bash`

where `ecstatic_sinoussi` is the running `gnome` and `drunk_sammet` is the running `squid`. Merging the containers IPC namespaces `--ipc:container:...` is necessary so that they can make sense of each others `/dev/shm` entries.

Why did we launch the container as root? So that we can install the appropriate drivers and customize our `/etc/machine-id`:

- `dnf -y install xorg-x11-drivers mesa-dri-drivers`
- `echo '257596e5b5324d1594cd856139ed4ed7' > /etc/machine-id`

where 257596e5b5324d1594cd856139ed4ed7 is the machine-id of the running `gnome` container. Once we've got the drivers installed, we're ready to become our user:

- `exec su tjay -`

And to align the pulseaudio settings with the `gnome` container:

- `mkdir -p ~/.config/pulse`
- `ln -s /tmp/pulse-YN0kDcYONUhZ ~/.config/pulse/257596e5b5324d1594cd856139ed4ed7-runtime`

where 257596e5b5324d1594cd856139ed4ed7 is the machine-id and pulse-YN0kDcYONUhZ is the pulse session name from earlier.

Now all that remains is:

- `export DISPLAY=":0"`
- `google-chrome`

Hide the Guake terminal with F12, and watch as Chrome opens in the GNOME desktop. If you visit: `chrome://gpu` you should see that full 3D acceleration is enabled! If you visit a site with sound, you should see that the GNOME mixer respects it as a normal application.

Just to avoid confusion, at you can also now stop the `upower` and `NetworkManager` services that GNOME started (since

they aren't touching *real* hardware). In a new Guake tab:

- `sudo systemctl stop upower`
- `sudo systemctl stop NetworkManager`

Overview of the Atomic Desktop

Before diving into some approaches to dealing with temperamental hardware, let's cover some of the implications.

Advantages and Possible Workflows

I can now (through Guake tabs themselves or screens) start up multiple docker containers from within my contained GNOME environment. I can make them full hardware accelerated peers (as I did with Chrome) or I can instead share X11 over a port and tell the containers about it via `--link` facilitating `export DISPLAY="x11:0"` or similar *just* working. Similarly, I can do the same for pulseaudio. I can share it as a first class peer or use the usual Docker networking setups.

I can "snapshot" my running `gnome` environment, I can freeze and unfreeze contained applications. With the squid cache, exploratory building is a snap. I certainly shouldn't be taken as a role model when it comes to workflows (I prefer the CLI over all else), but I would like to note a number of advantages.

Semi-Generic Containers for Most Applications

For almost all of my long running applications, I have a preprepared container. These include containers like:

- Chrome
- Firefox
- Libvirtd
- virsh
- mutt, postfix, fetchmail, procmail and recoll
- shell (this is my exploratory base including utilities like wget, mosh, vim, etc.)
- video (VLC, ffmpeg, ghb, etc.)

Notice how the Dockerfile for Chrome includes openvpn? This is so that it can be launched with `--privileged` and create it's own TUN device for just it to use. Almost all of my containers are similarly set to be launched in a number of different ways all emphasizing isolation. It use to always bother my that all of my DNS queries were going over my work VPN, now my routing can be per application.

A Working Set of Fresh Applications

Following the philosophy of installing nothing extra on the host, I also tend not to create Dockerfiles unless a task is nearly

omnipresent. By installing applications in a preprepared base just when I need them, I let the `squid` cache manage my hard drive space for me (currently as per the Dockerfile I've assigned it 50 gigs). Thus I automatically only pay longer install times for apps I use less. Further, because I pull them through the squid cache, I always use the latest version of applications with no extra effort on my part.

No Fear Updates

For applications I install just-in-time, I always am using the newest version. For apps that I choose to host in preprepared containers, updating is still easy and risk-free. All I need do is rebuild and tag with a testing postfix (for example `gnome` becomes `gnomep`). Then I can take the new `gnomep` container for a spin and if it works then "promote" it to `gnome`.

Immutable and Temporary by Default

My Desktop and Downloads folder used to constantly fill with disorganized sundry. Now by default all of my recent file working set lifecycles are tied to a container. I have to explicitly think about what I might want to move to the host (or alternatively if I want to make a snapshop of a container).

Now I tend to create a fresh container with "Developer Tools" and snapshot it for the duration of the exploratory portion of a project. Once things have become more organized, I simply commit the files the the host in an organized way (or to github) and then wipe the container images.

Explicit Configuration

Using the binding facility of Docker, It's easy to mix and match different configurations of a container at runtime. For example, here is a typical command to launch a shell:

- `sudo docker run -i -t -v /home/tjay/shared/configs/keys:/home/tjay/keys -v /home/tjay/shared/configs/prowl.py:/usr/bin/prowl.py -v /home/tjay/shared/configs/vimrc:/home/tjay/.vimrc --link furious_darwin:x11 --link reverent_hypatia:rpmcache -e 'DISPLAY=x11:17' -u tjay shell /bin/bash`

Both the keys directory and `prowl.py` contain sensitive information (API and SSH keys) but need not be housed in the container itself.

Hardware Specific Concerns

Poorly Supported Hardware

The procedures I've described so far work well on my laptop. However, at the office I---unfortunately---have an aging Dell Precision 380 as a workstation. This model has an Nvidia card requiring the outdated and spotily support 304 series of drivers. As I mentioned before, any container that wishes to make use of hardware acceleration will need access to a copy of the

appropriate drivers (though not necessarily kernel modules). To keep containers my containers as generic as possible, I've decided to leverage my RPM cache and always install the drivers at runtime. This is probably the easiest approach until such time as Docker supports mix-ins.

However, even this approach doesn't work completely for the Precision 380. For one, the default boot behavior loads an incompatible framebuffer driver. Second, the precompiled 304 drivers often lag behind the kernel available on Atomic. Third, unlike the Intel drivers, the Nvidia ones create a new device. Without getting into too much hardware specific detail, here is how I approached this issue:

- Replacing the normal Atomic `rhgb` boot option with `nomodeset` to prevent the loading of a graphics driver.
- using mknod to create the appropriate Nvidia devices on the host so that I can add them to containers (unfortunately, Docker has a limited ability to create devices or files that don't already exist on the host)
- Building an RPM build environment container that I use to build the akmod versions of the drivers for a given kernel. The "output" directory of this container is a shared volume that I import into my other containers, so I can install them at runtime.
- In addition to loading the shared drivers, I also issue manually issue an `insmod nvidia.ko` command in the `gnome` container before starting X. This sets up all of the needed kernel modules.

Dynamic Concerns

While the `gnome` container has access to the udev devices, it isn't actually running the udev service. The host is. This means that for certain reconfiguration events (such as docking) it is best to stop gnome, X, and the container and then restart them. I could run udev and power management inside a container, but I prefer not to as I sometimes want these features *before* I start Docker.

Conclusion

So far, I've experimented with running a fully contained desktop using both the minimal spin of Fedora and the Fedora flavor of Atomic with great success. I can share my Dockerfiles between my laptop and desktop to get a truly unified experience and I now have more control over how my applications work than ever.