Project Atomic Lab: End to End

<scollier@redhat.com>

Project Atomic Lab: End to End

<scollier@redhat.com>

Table of Contents

Chapter 1. Introduction

Project Atomic - geard, SELinux, Atomic, Docker, systemd. Project Atomic and Docker are very fast moving technologies at the moment. There is a already lot of existing documentation out there. This lab attempts to consolidate some of that documentation and provide a comprehensive example of an end to end deployment using Project Atomic components. The goal is to have a systems administrator deploy a Red Hat Enterprise Linux Atomic Host build server and compose some trees. Then, the admin would deploy a couple of Atomic Hosts and consume those trees. After that is done, we'll move into some Docker and geard basics and wrap up with a Red Hat Enterprise Linux Atomic host rollback. This should introduce you to all the components in the Project Atomic container ecosystem.

Chapter 2. Lab Prerequisites

2.1. Project Atomic Lab Pre-requisites

This lab was designed so that everything can be comleted on a single laptop.

1. Laptop Hardware
    a. Provisioned with Fedora 20 or RHEL 7
    b. Minimum 8GB RAM, 16GB preferred
    c. Minimum 50GB free space
2. Laptop Software
    a. Fedora 20 or RHEL 6/7
    b. Virtualization software
        i. qemu-kvm
        ii. virt-manager
        iii. libvirt-daemon-kvm
        iv. OR you can simply install the "Virtualization" group via "yum groupinstall Virtualization"
    c. The MongoDB client is required
3. Virtual Machines
    a. One virtual machine called: Atomic_Host_1
    b. One virtual machine called: Atomic_Host_2 (Only used for the multi-host / container linking geard lab)
    c. One virtual machine called: Compose_Server

Chapter 3. Red Hat Enterprise Linux Atomic Host Compose Server

3.1. Configure the Compose Server - As of 6/5/2014, this chapter is not complete. Skip.

The Atomic compose server builds the trees of content that the Red Hat Enterprise Linux Atomic Hosts consume.

3.2. Deploy a Red Hat Enterprise Linux 7 Virtual Machine

Note

Perform this on the hypervisor

1. Create the directory structure, from the *root* home directory

```
cd ~
mkdir -p atomic/build_server
```

2. Download the image

```
cd atomic/build_server
wget http://download.devel.redhat.com/brewroot/packages/rhel-server-kvm/7.0/7/images/r
```

3. Change the password on the image

```
virt-sysprep -a rhel-server-kvm-7.0-7.x86_64.qcow2 --root-password password:redhat
```

4. Install the Red Hat Enterprise Linux image

```
virt-install \
  --memory 4096 \
  --name Compose_Server \
  --disk ~/atomic/build_server/rhel-server-kvm-7.0-7.x86_64.qcow2 \
  --import \
  --noautoconsole
```

5. Open the VM virtual console

```
$ virsh
Welcome to virsh, the virtualization interactive terminal.

Type:  'help' for help with commands
       'quit' to quit
```

```
virsh # console Compose_Server
Connected to domain Compose_Server
```

6. Configure SSH

```
# sed -i 's/PasswordAuthentication no/PasswordAuthentication yes/' /etc/ssh/sshd_confi
# grep -i passwordauth /etc/ssh/sshd_config
#PasswordAuthentication yes
PasswordAuthentication yes
# PasswordAuthentication.  Depending on your PAM configuration,
# PAM authentication, then enable this but set PasswordAuthentication
# systemctl restart sshd
```

7. Get the IP address from the image and SSH in, instead of using the virt-console

8. Now you should be SSH'd into the VM. Add the collider repo for RHEL internal content

Note

Perform on the build server

```
cat << EOF > /etc/yum.repos.d/extras-rhel-7.repo
[extras-rhel-7-build]
name=RHEL 7 Extra Packages
baseurl=http://download.eng.bos.redhat.com/brewroot/repos/extras-rhel-7.0-build/latest/x8
enabled=1
gpgcheck=0
EOF
```

1. Install *rpm-ostree* tools

```
yum -y install httpd yum-utils ostree binutils nss-altfiles firewalld
```

2. Edit the */etc/nsswitch.conf* file

```
sed -i 's/passwd:     files sss/passwd:     files sss altfiles/' /etc/nsswitch.conf
sed -i 's/shadow:     files sss/passwd:     files sss altfiles/' /etc/nsswitch.conf
```

3. Disable SELinux on the compose server

```
sed -i 's/^SELINUX=.*/SELINUX=disabled/g' /etc/selinux/config && cat /etc/selinux/conf
```

4. Create the repo and set up Apache

```
mkdir /srv/rpm-ostree &&
cd /srv/rpm-ostree &&
mkdir -p repo &&
ostree --repo=repo init --mode=archive-z2 &&

cat > /etc/httpd/conf.d/rpm-ostree.conf <<EOF
DocumentRoot /srv/rpm-ostree
<Directory "/srv/rpm-ostree">
Options Indexes FollowSymLinks
AllowOverride None
Require all granted
</Directory>
EOF
```

5. Create a systemd service file

```
cat > /etc/systemd/system/rpm-ostree-autobuilder.service <<EOF
[Unit]
Description=RPM-OSTree autobuilder

[Service]
WorkingDirectory=/srv/rpm-ostree
ExecStart=/usr/bin/rpm-ostree-autobuilder

[Install]
WantedBy=multi-user.target
EOF
```

6. Enable services and configure firewall

```
systemctl start firewalld &&
systemctl enable firewalld &&
systemctl daemon-reload &&
systemctl enable httpd &&
systemctl start httpd &&
systemctl reload httpd &&
systemctl enable rpm-ostree-autobuilder &&
systemctl start rpm-ostree-autobuilder &&
firewall-cmd --add-service=http &&
firewall-cmd --add-service=http --permanent
```

7. Reboot the host

```
systemctl reboot
```

8. Create the *products.json* file

```
cat > /srv/rpm-ostree/products.json << EOF
{
    "comment": "Red Hat Enterprise Linux Atomic Host 7.0",

    "osname": "rhel-atomic-host",
    "ref": "rhel-atomic-host/7.0-buildmaster/x86_64/base",

    "repos": ["extras-rhel-7-build", "extras-rhel-7-candidate"],

    "selinux": true,

    "bootstrap_packages": ["filesystem", "glibc", "nss-altfiles", "shadow-utils",
         "redhat-release-atomic-controller"],

    "packages": ["kernel", "rpm-ostree", "lvm2", "syslinux-extlinux",
     "btrfs-progs", "e2fsprogs", "xfsprogs",
     "docker",
     "selinux-policy-targeted",
     "audit",
     "min-cloud-agent",
     "subscription-manager",
     "openssh-server", "openssh-clients",
     "passwd",
     "NetworkManager", "vim-minimal", "nano",
     "sudo"],

    "units": ["docker.service", "docker.socket"]
}
```

```
EOF
```

FIXME NEED PACKAGE FROM COLLIDER BEFORE COMPLETING THIS SECTION

3.2.1. Links

http://www.projectatomic.io/blog/2014/04/build-your-own-atomic-host-on-fedora-20/

Chapter 4. Configure Project Atomic VMs

4.1. Options for Deployment

There are a few different ways to obtain Atomic images:

1. In .qcow2 format
2. Deploy from rpm

For the purposes of this lab, we will be deploying Red Hat Enterprise Linux Atomic Controllers via the .qcow2 pre-built images.

4.2. Obtaining the images

1. On the host running the VM's, create the directory structure.

```
cd ~
mkdir -p /var/lib/libvirt/images/atomic/{1,2}
```

2. Download and decompress the images to the appropriate directories. You only need to do the following step if the images haven't been downloaded for you already. Perform this as a user with *sudo* access.

```
cd ~
pushd /var/lib/libvirt/images/atomic/1/
wget http://rcm-img06.build.bos.redhat.com/images/auto/rh-atomic-controller-el7-x86_64
xz -d latest-qcow2.xz
cp latest-qcow2 ../2/
popd
```

3. Install the image and create an additional drive

```
cd ~
for i in 1 2; do
  sudo virt-install \
  --memory 2048 \
  --name Atomic_Host_$i \
  --disk /var/lib/libvirt/images/atomic/$i/latest-qcow2 \
  --import \
  --noautoconsole \
  --disk /var/lib/libvirt/images/atomic/$i/storage_$i.qcow2,format=qcow2,sparse=true,s
done
```

4. On each Atomic host, launch the console. Set a password for the *root* user on each Atomic host. By default, the image does not have a password set for the root user. So, log in without a password and set it. In addition, these images do not have a serial console. Use a graphical console, either through *virt-manager* or *virt-viewer*.

```
virt-viewer Atomic_Host_1
passwd
```

5. Get IP Address of both hosts and open up a terminal console to each machine.

4.3. Configure Disk Partitions

The Red Hat Enterprise Linux Atomic Controllers come with a fairly small disk image. We need to confiure the second disk that was added with *virt-install* and mount that under */var/lib/docker*.

Note

Ensure you are on the correct host! It would not be good if you are on the wrong host. You should be on the Atomic VM's when you perform the following operation.

1. Create the Atomic_Host_sdb.layout file that will be used to partition *sdb*. You can just copy the following contents into your shell on the Atomic Host VM's.

```
cat << EOF > Atomic_Host_sdb.layout
# partition table of /dev/sdb
unit: sectors

/dev/sdb1 : start=     2048, size= 20969472, Id=83
/dev/sdb2 : start=        0, size=        0, Id= 0
/dev/sdb3 : start=        0, size=        0, Id= 0
/dev/sdb4 : start=        0, size=        0, Id= 0
EOF

sfdisk /dev/sdb < Atomic_Host_sdb.layout
```

2. Format */dev/sdb1*

```
mkfs.xfs /dev/sdb1
```

3. Ensure that only */dev/sdb1* is mounted to */var/lib/docker*

```
systemctl stop docker
mount
df -hal
umount /var/lib/docker
mount /dev/sdb1 /var/lib/docker
mount
```

4. Get the UUID from */dev/sdb1* and add it to the */etc/fstab*

```
cp /etc/fstab{,.orig}
cat /etc/fstab
echo "$(blkid /dev/sdb1 | awk '{ print $2 }') /var/lib/docker xfs defaults 1 1" >> /et
cat /etc/fstab
mount -a
mount
df -hal
systemctl start docker
ls /var/lib/docker
```

4.4. Validate Hostname and Networking

1. Set the hostname for each of your Red Hat Enterprise Linux Atomic Controller's.

```
hostnamectl set-hostname atomic1.local     # on host 1
```

```
hostnamectl set-hostname atomic2.local     # on host 2
```

2. Ensure you can ping the other controller

```
ping 10.0.0.1     # from host 1, replace IP address for host 2
```

3. Ensure each host can resolve DNS

```
host redhat.com     # from each host
```

4.5. Validate SELinux

1. Ensure that selinux is set to enforcing mode on each controller.

```
getenforce
```

4.6. Ensure Docker and geard are Functional

```
systemctl status docker
systemctl status geard
```

4.7. Upgrade both controllers.

1. Disable GPG support

```
echo "gpg-verify=false" >> /ostree/repo/config
```

2. Run the following on both of the Atomic controllers.

```
rpm-ostree upgrade
reboot
```

Chapter 5. Building Docker Images

5.1. Overview

In this section we will build images that will be used through the remainder of the lab. The Dockerfiles will be provided in the text below, just copy and paste them with the appropriate filenames. The Dockerfiles in this section are just examples. Some are Fedora, some are RHEL. Try them both out.

This section describes building the images, but doesn't go into detail on the usage. That will be in the following chapters.

1. Download the Dockerfiles and Scripts to each Atomic host and extract them.

```
curl -O https://raw.githubusercontent.com/scollier/project-atomic-lab-end-to-end/maste
tar -xvf dockerfiles.tar
```

5.1.1. Apache Image Atomic Host 1

Build this image on Atomic Host 1

```
docker images
cd ~/Dockerfiles/atomic_host/apache
docker build -t demo/apache .
```

5.1.2. Apache Image Atomic Host 2

Build this image on Atomic Host 2

```
docker images
cd ~/Dockerfiles/atomic_host/apache
docker build -t demo/apache .
```

5.1.3. Mediawiki Image

Build this image on Atomic Host 1. Build the image

```
docker images
cd ~/Dockerfiles/atomic_host/mediawiki
docker build -t demo/mediawiki .
```

5.1.4. MariaDB Image

Build this image on Atomic Host 1. Build the image

```
docker images
cd ~/Dockerfiles/atomic_host/mariadb
docker build -t demo/mariadb .
```

5.1.5. nginx Image

Build this image on Atomic Host 1 . Build the image

```
docker images
cd ~/Dockerfiles/atomic_host/nginx
docker build -t demo/nginx .
```

5.1.6. MongoDB Image

Build this image on Atomic Host 1. Build the image

```
docker images
cd ~/Dockerfiles/atomic_host/mongodb
docker build -t demo/mongodb .
```

5.1.7. Links

More example Dockerfiles are on the Fedora Cloud SIG github page:

https://github.com/fedora-cloud/Fedora-Dockerfiles

Chapter 6. Docker Basics

6.1. Overview

The rapid adoption of Docker demonstrates that the benefits of Docker and containers in general are valued by enterprise developers and administrators. Specifically, Docker and containers enable rapid application deployment by only including the minimal runtime requirements of the application. This minimal size and the mentality of replacing containers, rather than updating them, simplifies maintenance. Additionally, containers allow applications bring all of their runtime requirements with them, making them portable across multiple Red Hat Enterprise Linux environments. This means that containers can ease testing and troubleshooting efforts by providing a consistent runtime across development, QA and production environments. In addition, containers run applications in isolated memory, process, filesystem and networking spaces. The isolation ensures that any security breaches are limited to the container.

Red Hat has been investing in containers for a number of years in Red Hat Enterprise Linux and has been working on Docker in the upstream community since mid-2013. Red Hat's commitment to Docker and container technology is demonstrated not just in this background work, but also in the efforts to establish Docker containers as a standard part of the Red Hat Enterprise Linux environment. Red Hat has production experience leveraging container technologies like cgroups and namespaces since Red Hat Enterrise Linux 6. Establishing, consuming and sharing these capabilities as a part of Red Hat Enterprise Linux is a major step in making them consumable by enterprise customers.

This lab has 3 sections:

1. Overview

2. Lab 1: Docker Environment

3. Lab 2: Containers can Talk

6.1.1. What you can expect to learn from this lab

Topics covered:

1. Explore Docker

2. Saving Content

3. Host exploration

4. External Logging

5. Starting containers on boot

6. Linking containers

6.1.2. Lab Environment

Note

Perform all activities on the Red Hat Enterprise Linux Atomic Host 1 unless instructed otherwise.

1. SSH Access

```
ssh root@IPAddress_of_RHELAH1
```

6.2. Docker Overview

6.2.1. Explore Docker

All actions in this lab will performed by a user with root privileges.

1. Check to ensure that SELinux is running on the host.

```
getenforce
```

2. Take a look at the documetation and general help as well as command specific help that is provide by the Docker package.

```
rpm -qd docker
man docker
man docker-run
docker --help
docker run --help
```

3. A Docker *image* is basically a layer. A layer never changes. Take a look at the images that are on this system. There should be *nginx*,*mariadb*,mongodb,mediawiki,apache.

```
docker images --help
docker images
```

4. Docker provides the *run* option to run a image. Check out the *run* options and then run the image. The following command launches the image, executes the command *echo hello*, and then exits.

```
docker run --help
docker run fedora echo hello
```

5. Check the logs. The following commands will list the last container that ran so you can get the UUID and check the logs. This should return the output of "echo hello". Finally, run with the *-t* option to allocate a psuedo-tty. Note that *-l* below is lowercase *L*.

```
docker ps -l
docker logs <Container UUID>
```

6. Tag the base image with a new name.

```
docker tag registry.access.redhat.com/redhat/rhel7beta rhel7
```

```
docker images | grep rhel7
```

7. To run an interactive instance that you can look around in, pass the options *-i* and *-t*. The *-i* option starts an interactive terminal. The *-t* option allocates a pseudo-tty. You should see different results than before.

```
docker run -i -t rhel7 bash
```

8. Explore the *hosts* file and look at the IP Address. Try some *ip* commands… This is a slimmed down base image. There are not many tools on here by default. Exit the container when finished.

```
cat /etc/hosts
ip a
ip r
exit
```

9. Switch to the Fedora image and take a look around. Grab the hostname of the container. By default the hostname is set to the UUID of the container. We will look at how to change that later.

```
docker run -i -t fedora bash
ip a
ip r
hostname
```

10. What processes are running inside the container?

```
ps aux
```

11. What is the SELinux security label of the processes?

```
ps -Z
```

6.2.2. Saving Content

Note

Perform these actions while still inside the Fedora container.

Now that we have an idea of what's going on inside the container, let's take a look at the process required to save a file.

1. Create a file inside the container and see if it persists the next time you run the container.

```
echo "Hello World" >> ~/file1
ls ~/
```

2. Exit the container.

```
exit
```

3. Run the container again and check to see if the file exists. The file should be gone.

```
docker run -i -t fedora bash
ls ~/
```

4. Let's try this again and this time we'll commit the container.

```
echo "Hello World" >> ~/file2
```

5. Exit the container and commit the container.

```
exit
docker ps -l
docker commit <Container UUID> file2/container
ae4b621fc73d0a66bf1e98657dee570043cb7f9910c0b96782a914fee85437f2
```

6. Now lets see if it saved the file. Now *docker images* should show the newly commited container. Launch it again and check for the file.

```
docker images
docker run -i -t file2/container bash
ls ~/
exit
```

6.2.3. Host exploration

Now that we have explored what's on the inside of a container, let's see what is going on outside of the container.

1. Let's launch a container that will run for a long time then confirm it is running. The *-d* option runs the container in daemon mode. Remember, you can always get help with the options. Run these commands on the host (you should not be inside a container at this time).

```
docker run --help
docker run -d rhel7 sleep 999999
```

2. List the images that are currently running on the system.

```
docker ps
```

3. Now, check out the networking on the host. You should see the *docker0* bridge and a *veth* interface attached. The *veth* interface is one end of a virtual device that connects the container to the host machine. You should see that the IP address of the bridge is used as the default gateway of the container that you saw earlier.

```
ip a
```

4. What are the firewall rules on the host? You can see from the *nat* table that all the traffic is masqueraded so that you can reach the outside world from the containers.

```
iptables -nvL
iptables -nvL -t nat
```

5. What is Docker putting on the file system? Check */var/lib/docker* to see what Docker actually puts down.

```
ls /var/lib/docker
```

6. The root filesystem for the container is in the devicemapper directory. Grab the *Container ID* and complete the path below. Replace <Container UUID> with the output from *docker ps -l* and use tab completion to complete the <Container UUID>.

```
docker ps -l
```

```
ls /var/lib/docker/devicemapper/mnt/<Container ID><tab><tab>/rootfs
```

7. How do I get the IP address of a running container? Grab the <Container UUID> of a running container.

```
docker ps
docker inspect <Container UUID>
```

8. That is quite a lot of output, let's add a filter. Replace <Container ID> with the output of *docker ps*.

```
docker ps
docker inspect --format '{{ .NetworkSettings.IPAddress }}' <Container UUID>
```

9. Stop the container and check out its status. The container will not be running anymore, so it is not visible with *docker ps*. To see the <Container ID> of a stopped container, use the *-a* option. The *-a* option shows all containers, started or stopped.

```
docker stop <Container UUID>
docker ps
docker ps -a
```

6.2.4. Where are my logs?

The containers do not run syslog. In order to get logs from the container, there are a couple of methods. The first is to run the container with */dev/log* socket bind mounted inside the container. The other is to write to external volumes. That's in a later lab.

1. Launch the container with an interactive shell. The file */dev/log* is a socket.

```
docker run -v /dev/log:/dev/log -i -t rhel7 bash
```

2. Now that the container is running. Open another terminal and inspect the bind mount. Do not run this inside the container.

```
docker ps -l
docker inspect --format '{{.Volumes}}' <Container UUID>
```

3. Go back to the original terminal. Generate a message with *logger* and exit the container. This should write the message to the host journal.

```
logger "This is a log Entry"
exit
```

4. Check the logs on the host to ensure the bind mount was successful.

```
journalctl | grep -i "This is a log Entry"
```

6.2.5. Control that Service!

We can control services with systemd. Systemd allows us to start, stop, and control which services are enabled on boot, among many other things. In this section we will use systemd to enable the *nginx* service to start on boot.

1. Have a look at the docker images.

```
docker images
```

2. You will notice a repository called *demo/nginx*, that is what will be used in this section.

3. Here is the systemd unit file that needs to be created in order for this to work. The content below needs to be placed in the */etc/systemd/system/nginx.service* file. This is a trivial file that does not provide full control of the service.

```
cat > /etc/systemd/system/nginx.service << EOF
[Unit]
Description=nginx server
After=docker.service

[Service]
Type=simple
ExecStart=/bin/bash -c '/usr/bin/docker start nginx || /usr/bin/docker run --name ngin

[Install]
WantedBy=multi-user.target
EOF
```

4. Now control the service. Enable the service on reboot.

```
systemctl enable nginx.service
systemctl is-enabled nginx.service
```

5. Start the service. When starting this service, make sure there are no other containers using port 80 or it will fail.

```
docker ps
systemctl start nginx.service
docker ps
```

It’s that easy!

1. Before moving to the next lab, ensure that *nginx* is stopped, or else there will be a port conflict on port 80.

```
docker ps | grep -i nginx
```

2. If it is running:

```
docker stop nginx
systemctl disable nginx.service
```

6.3. Containers can Talk

Now that we have the fundamentals down, let’s do something a bit more interesting with these containers. This lab will cover launching a *MariaDB* and *Mediawiki* container. The two will be tied together via the Docker *link* functionality. This lab will build upon things we learned in lab 1 and expand on that. We’ll be looking at external volumes, links, and additional options to the Docker *run* command.

A bit about links

Straight from the Docker.io site:

"Links: service discovery for docker. Links allow containers to discover and securely communicate with each other by using the flag -link name:alias When two containers are linked together Docker creates a parent child relationship between the containers. The parent container will be able to access information via environment variables of the child such as name, exposed ports, IP and other selected environment variables."

6.4. MariaDB

This section shows how to set up an external volume and use hostnames when launching the MariaDB container.

6.4.1. Review the MariaDB Environment

1. Review the scripts and other content that are required to build and launch the *MariaDB* container. This lab does not require that you build the container as it has already been done to save time. Rather, it provides the information you need to understand what the requirements of building a container like this.

```
cd /root/Dockerfiles/atomic_host_1/mariadb/; ls
```

2. Review the Dockerfile. Look at the *Dockerfile*. From the contents below, you can see that the Dockerfile is starting with the Fedora base image and is maintained by Scott Collier. After the *FROM* and *MAINTAINER* commands are run, the commands to install software are run with *RUN*. Think of the *RUN* command as executing a line in a shell script. After the software is installed we do some configuration of MariaDB. Next, we *ADD* a basic MariaDB configuraiton file as well as a script that will be used to do more configuration and launch the database. Finally *EXPOSE* and *CMD* which expose ports and provide the starting command, respectively. Exposing the port will make the port available to the *Mediawiki* container when it is launched with the *-link* command.

```
MAINTAINER Scott Collier <scollier@redhat.com>

RUN yum -y install mariadb-server pwgen supervisor psmisc net-tools; yum clean all

RUN mkdir -p /var/log/mysql && \
    touch /var/log/mysql/.keep /var/lib/mysql/.keep && \
    chown -R mysql:mysql /var/log/mysql /var/lib/mysql

ADD ./simple.cnf /etc/my.cnf.d/
ADD ./config_mariadb.sh /config_mariadb.sh

EXPOSE 3306

CMD [ "/config_mariadb.sh" ]
```

3. Review the *simple.cnf* file. This configuraiton file has some basic settings for how we want to run the MariaDB container.

```
[client]
default-character-set  = utf8

[mysqld_safe]
nice    = 0
```

```
log-error=/var/log/mysql/mysqld.log

[server]
# user    = mysql
user    = root
tmpdir    = /tmp
skip-external-locking

max_connections   = 32
connect_timeout   = 5
wait_timeout    = 600
max_allowed_packet  = 16M
thread_cache_size       = 128
sort_buffer_size  = 4M
bulk_insert_buffer_size = 16M
tmp_table_size    = 32M
max_heap_table_size = 32M
myisam_recover          = BACKUP
key_buffer_size   = 128M
table_cache             = 400
myisam_sort_buffer_size = 512M
concurrent_insert = 2
read_buffer_size  = 2M
read_rnd_buffer_size  = 1M
query_cache_limit   = 128K
query_cache_size    = 64M
log_warnings    = 2
slow_query_log
slow_query_log_file   = /var/log/mysql/mariadb-slow.log
long_query_time = 10
log_slow_verbosity  = query_plan
log_slow_admin_statements
log_bin     = /var/log/mysql/mariadb-bin
log_bin_index   = /var/log/mysql/mariadb-bin.index
expire_logs_days  = 10
max_binlog_size         = 100M
default_storage_engine  = InnoDB
sql_mode    = NO_ENGINE_SUBSTITUTION,TRADITIONAL
innodb_log_file_size  = 16M
innodb_buffer_pool_size = 265M
innodb_log_buffer_size  = 8M
innodb_file_per_table = 1
innodb_open_files = 400
innodb_io_capacity  = 400
innodb_flush_method = O_DIRECT

[mysqldump]
quick
quote-names
max_allowed_packet  = 16M

[mysql]

[isamchk]
key_buffer    = 16M
```

4. Review the config_mariadb.sh file. This script installs the database, sets permissions on some directories, creates a test database and assigns the appropriate access controls. Of course, this is just an example and it can be heavily modified.

```
# cat config_mariadb.sh
#!/bin/bash

__mysql_config() {
mysql_install_db
mkdir -vp /var/run/mariadb
chown -vR mysql:mysql /var/run/mariadb/
chown -R mysql:mysql /var/lib/mysql/
chown -R mysql:mysql /var/log/mariadb/
cd '/usr' ; /usr/bin/mysqld_safe --datadir='/var/lib/mysql' &
sleep 10

echo "Running the start_mysql function."
mysqladmin -u root password mysqlPassword
mysql -uroot -pmysqlPassword -e "UPDATE mysql.user SET Password=PASSWORD('mysqlPasswor
mysql -uroot -pmysqlPassword -e "GRANT ALL PRIVILEGES ON testdb.* TO 'testdb'@'localho
mysql -uroot -pmysqlPassword -e "GRANT ALL PRIVILEGES ON *.* TO 'testdb'@'%' IDENTIFIE
mysql -uroot -pmysqlPassword -e "delete from user where user='';
mysql -uroot -pmysqlPassword -e "GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED
mysql -uroot -pmysqlPassword -e "select user, host FROM mysql.user WHERE Host <> 'loca
sleep 10
killall mysqld
rm -f /var/lib/mysql/mysql.lock
exec /usr/libexec/mysqld
}

# Call all functions
__mysql_config
```

6.4.2. Launch the MariaDB Container

1. Either tail the audit log from your current terminal by placing the tail command in the background:

   ```
   tail -f /var/log/audit/audit.log | grep -i avc &
   ```

2. Or open another terminal and watch for AVCs in the foreground:

   ```
   tail -f /var/log/audit/audit.log | grep -i avc
   ```

3. Launch the container. The /mariadb/db directory already exists and has database content inside.

   ```
   docker run -p 3306:3306 -t -d --name mariadb -v /mariadb/db:/var/lib/mysql -v /mariadb
   ```

4. Did the container start as expected? You should see some AVC's. Look at the logs on the container and see the *permission denied* messages.

   ```
   docker logs mariadb
   ```

5. You will need to allow the proper SELinux permissions on the local */mariadb/db* directory so *MariaDB* can access the directory. Right now it's at *default_t*, this needs to be changed per below. In addition, create the directory that we will use for the MariaDB log files and change it's SELinux permissions as well.

   ```
   ls -lZd /mariadb/db
   chcon -Rvt svirt_sandbox_file_t /mariadb/db/
   ls -lZd /mariadb/db
   mkdir -vp /mariadb/logs
   ```

```
chcon -Rvt svirt_sandbox_file_t /mariadb/logs/
```

6. Whether or not the container is still running, you will have to be removed because of a naming conflict.

```
docker ps -a
docker stop mariadb && docker rm mariadb
```

7. Grab the database files for the lab and extract them. The purpose of this is to create an environment that already exists. By using these database files, the wiki will start right up. Otherwise, you could just create the wiki from scratch.

```
cd /mariadb/db/
curl -O https://raw.githubusercontent.com/scollier/project-atomic-lab-end-to-end/maste
tar xvf database_files.tar
cd
```

8. Launch the container again.

```
docker run -p 3306:3306 -t -d --name mariadb -v /mariadb/db:/var/lib/mysql -v /mariadb
docker ps -l
docker logs mariadb
```

9. Take a look at the database files in */mariadb/db/* and also take a look at the log files in */mariadb/logs/*.

```
ls /mariadb/db/*
tail -f /mariadb/logs/mariadb.log
```

The container should be running at this time. Kill the background *tail -f* process if you want.

6.5. Mediawiki

This section shows how to launch the *Mediawiki* container and link it back to the *MariaDB* container.

6.5.1. Review the Mediawiki Environment

Review the scripts and other content that are required to build and launch the *Mediawiki* container and link it to the *MariaDB* container. This lab does not require that you build the container as it has already been done in the prior lab. Rather, it provides the information you need to understand what the requirements of building a container like this. The files are pasted here, but they are also in */root/Dockerfiles/atomic_host_1/mediawiki*

1. Review the Dockerfile. This is a systemctl based dockerfile. There is work being done so that the *wants* files don’t have to be removed. At some point there will be a minimal systemd environment for containers.

```
# cat Dockerfile
FROM fedora:20
MAINTAINER Scott Collier <scollier@redhat.com>

ENV container docker

VOLUME [ "/sys/fs/cgroup" ]
```

```
RUN yum -y update; yum clean all
RUN yum -y install systemd mediawiki php php-mysqlnd httpd; yum clean all; \
(cd /lib/systemd/system/sysinit.target.wants/; for i in *; do [ $i == systemd-tmpfiles
rm -f /lib/systemd/system/multi-user.target.wants/*;\
rm -f /etc/systemd/system/*.wants/*;\
rm -f /lib/systemd/system/local-fs.target.wants/*; \
rm -f /lib/systemd/system/sockets.target.wants/*udev*; \
rm -f /lib/systemd/system/sockets.target.wants/*initctl*; \
rm -f /lib/systemd/system/basic.target.wants/*;\
rm -f /lib/systemd/system/anaconda.target.wants/*; \
systemctl enable httpd.service
# yum -y erase iprutils

# Now wiki data.  We'll expose the wiki at $host/wiki, so the html root will be
# at /var/www/html/wiki; to allow this to be used as a data volume we keep the
# initialisation in a separate script.

ADD ./config.sh /config.sh
ADD ./LocalSettings.php /var/www/html/wiki/
RUN chmod +x /config.sh
RUN /config.sh

ADD run-mw.sh /run-mw.sh
RUN chmod +x /run-mw.sh

EXPOSE 80

CMD [ "/run-mw.sh" ]
```

2. Review the config.sh script. Check the comments in the script.

```
#!/bin/bash
#
# The mediawiki rpm installs into /var/www/wiki.  We need to symlink this into
# the served /var/www/html/ tree to make them visible.
#
# Standard config will put these in /var/www/html/wiki (ie. visible at
# http://$HOSTNAME/wiki )

mkdir -p /var/www/html/wiki

cd /var/www/html/wiki
ln -sf ../../wiki/* .

# We want /var/www/html/wiki to be usable as a data volume, so it's
# important that persistent data lives here, not in /var/www/wiki.

chmod 711 .
rm -f images
mkdir images
chown apache.apache images
```

3. Review the run-mw.sh script. Check the comments in the script.

```
#!/bin/bash
#
# Run mediawiki in a docker container environment.
```

```
# If we are talking to a mariadb/mysql instance in a linked container
# (aliased "db" on port 3306), then we need to dynamically update the
# MW config to refer to the correct DB server IP address.
#
# Docker will set the DB_PORT_3306_TCP_ADDR env variable to the right
# IP in this case.
#
# We'll update lines like
#   $wgDBserver = "localhost";
# to point to the correct location.

if [ "x$DB_PORT_3306_TCP_ADDR" != "x" ] ; then
    # For initial configuration, it's also considerate to update the
    # default settings that drive the config screen defaults
    sed -i 's/^\$wgDBserver =.*$/\$wgDBserver = "'$DB_PORT_3306_TCP_ADDR'";/' /usr/sha

    # Only update LocalSettings if they already exist; on initial
    # setup they will not yet be here
    if [ -f /var/www/html/wiki/LocalSettings.php ] ; then
    sed -i 's/^\$wgDBserver =.*$/\$wgDBserver = "'$DB_PORT_3306_TCP_ADDR'";/' /var/www
    sed -i 's/^\$wgServer =.*$/\$wgServer = "http:\/\/'$HOST_IP'";/' /var/www/html/wik
    fi
fi

# Finally fall through to the apache startup script that the apache
# Dockerfile (which we build on top of here) sets up

exec /usr/sbin/init
```

6.5.2. Launch the Mediawiki Container

This section shows how to use hostnames and link to an existing container. Issue the *docker run* command and link to the *mariadb* container.

Run the container. The command below is taking the enviroment variable *HOST_IP* and will inject that into the *run-mw.sh* script when the container is launched. The *HOST_IP* is the IP address of the virtual machine that is hosting the container. Replace IP_OF_VIRTUAL_MACHINE with the IP address of the virtual machine running the container.

Note

In the following command, after the *-e*, leave the *HOST_IP* entry. It's used to hold the variable of the IP address of the Atomic Host 1 virtual machine.

```
ip a

docker run \
-e=HOST_IP=IP_OF_VIRTUAL_MACHINE \
--link mariadb:db \
-v /var/www/html/ \
--name mediawiki \
--privileged \
-p 80:80 \
-t \
-d \
-e 'container=docker' \
-v /sys/fs/cgroup:/sys/fs/cgroup:ro demo/mediawiki
```

1. Explore the link that was made.

```
docker ps | grep media
```

Note

Notice in the *NAMES* column on the mariadb container and how the link is represented.

1. Inspect the container and get volume information:

```
docker inspect --format '{{ .Volumes }}' mediawiki
```

2. Now take the output of the *docker inspect* command and use the UUID from that in the next command. Explore the mediawiki content. This directory is mapped to */var/www/html/wiki* inside the container.

```
ls /var/lib/docker/vfs/dir/<UUID Listed from Prior Query>/wiki
```

1. For example, see how the *LocalSettings.php* file is there and has the correct content:

```
ls /var/lib/docker/vfs/dir/1c8c23c24ebaea8e00fb8639e545c662516445faee7dcd5d89882fdbf1f
```

2. Open browser on the host running the VM and confirm the configuration is complete.

```
firefox &
```

3. Go to the *Mediawiki* home page. Use the IP address of the virtual machine. The same IP address that was passed in as the HOST_IP in the docker run command.

```
http://ip.address.here/wiki
```

4. Thats it. Now you can start using your wiki. You can click on *Create Account* in the top right and test it out, or log in with:

```
Username: admin
Passwrod: password
```

5. Now, how did this work? The way this works is that the Dockerfile *CMD* command tells the container to launch with the *run-mw.sh* script. Here's the key thing about what that script is doing, let's review:

```
if [ "x$DB_PORT_3306_TCP_ADDR" != "x" ] ; then
    # For initial configuration, it's also considerate to update the
    # default settings that drive the config screen defaults
    sed -i 's/^\$wgDBserver =.*$/\$wgDBserver = "'$DB_PORT_3306_TCP_ADDR'";/' /usr/sha

    # Only update LocalSettings if they already exist; on initial
    # setup they will not yet be here
    if [ -f /var/www/html/wiki/LocalSettings.php ] ; then
      sed -i 's/^\$wgDBserver =.*$/\$wgDBserver = "'$DB_PORT_3306_TCP_ADDR'";/' /var/w
      sed -i 's/^\$wgServer =.*$/\$wgServer = "http:\/\/'$HOST_IP'";/' /var/www/html/w
    fi
fi
```

It's doing a check for an existing LocalSettings.php file. We added that file during the Docker build process. That file was copied to /var/www/html/wiki. So, the script runs, sees that the file exists and points the *$wbDBserver* variable to the MariaDB container. So, no matter if these containers get shut

down and have new IP addresses, the Mediawiki container will always be able to find the MariaDB container because of the *link*. In addition, it's using the *-e* option to pass environment variables, in this case, $HOST_IP to the *run-mw.sh* script to complete the configuration.

Note

Stop and delete the mediawiki and mariadb containers before moving forward.

Chapter 7. Introduction to geard

7.1. geard

geard is a Docker container orchestration tool. At the currrent release, it essentially does three things:

1. SSH
2. Multiple container deployment
3. Link

This lab has 5 parts.

1. Single host / single container deployment
2. Single host / multi container deployment
3. Single host / MongoDB replica set configuration
4. Multi host container linking
5. SSH enablement for contianers

7.2. geard Lab Prerequisites

1. Two Atomic hosts

```
hostname   # on host 1
hostname   # on host 2
```

2. Copy down the json files to both hosts. Open a terminal session to each host and complete the following.

```
cd

for i in http_single.json mongo_deploy.json mongo_replica.json; do
  curl -O https://raw.githubusercontent.com/scollier/project-atomic-lab-end-to-end/mas
done
```

3. Check to see that geard and Docker are installed and running

```
rpm -qa | grep -i geard
rpm -qa | grep -i docker
systemctl status geard
systemctl status docker
```

4. Check for the proper .json files in the users home directory

```
ls *.json
http_single.json  mongo_deploy.json  mongo_replica.json
```

7.3. Deploy a Single Container

On Atomic Host 1:

1. Check which units geard has registered

```
gear list-units
```

2. Check to see which images are available and running within Docker

```
docker images
docker ps
```

3. Install the first unit

```
gear install demo/mongodb mongodb --start -p 27017:27017
```

4. List the units again

```
gear list-units
```

5. Show the container is also recognized by Docker

```
docker ps
```

6. Make sure the MongoDB server is responding to requests. It will notify you that you are trying to access the database over the native driver port. Just ignore it, we are just making sure it is listening.

```
curl http://localhost:27017
```

7. List the units one more time

```
gear list-units
```

8. Clean up the environment

```
gear delete mongodb
gear list-units
docker ps
```

7.4. Deploy a MongoDB replica set on a single host

On host 1:

1. Check the environment

```
gear list-units
docker ps
```

2. Explore the *mongo_deploy.json*, notice the name, count and image. The .json file is also taking care of the linking.

```
cd
```

```
cat mongo_deploy.json
{
  "containers":[
    {
      "name":"db",
      "count":3,
      "image":"demo/mongodb",
      "publicports":[
        {"internal":27017,"external":0}
      ],
      "links":[
        {"to":"db","nonlocal":true,"matchport":true}
      ]
    }
  ]
}
```

3. Deploy the application

```
# gear deploy mongo_deploy.json
==> Deploying mongo_deploy.json
ports: searching block 41, 4000-4099
ports: Reserved port 4000
local PortMapping: 27017 -> 4000
local Container db-1 is installed
ports: Reserved port 4001
local PortMapping: 27017 -> 4001
local Container db-2 is installed
ports: Reserved port 4002
local PortMapping: 27017 -> 4002
local Container db-3 is installed
==> Linking db: 192.168.1.1:27017 -> localhost:4000
==> Linking db: 192.168.1.2:27017 -> localhost:4001
==> Linking db: 192.168.1.3:27017 -> localhost:4002
local Container db-1 starting
local Container db-2 starting
local Container db-3 starting
==> Deployed as mongo_deploy.json.20140605-203024
```

4. Copy the contents of the replica set file and paste that into the mongodb shell, which you will be launching in the next step.

```
cat ~/mongo_replica.json
cfg = {
    "_id" : "replica0",
    "version" : 1,
    "members" : [
        {
            "_id" : 0,
            "host" : "192.168.1.1:27017"
        },
        {
            "_id" : 1,
            "host" : "192.168.1.2:27017"
        },
        {
            "_id" : 2,
            "host" : "192.168.1.3:27017"
```

```
        },
    ]
}
```

5. List the units and container

```
gear list-units
docker ps
```

6. This lab does require that you have the MongoDB client installed on a workstation that can access this VM. Connect with the MongoDB client. Install the *mongodb* package. The PORT that you are connecting to on the next step is the port that *geard* mapped. Most likely 400x. Replace *IP_OF_VM* with the IP address of the host running the MongoDB replica set. This should the IP Address Atomic Host 1. Connect to the first MongoDB server, this should be running on port 4000.

```
mongo --host IP_OF_VM --port "PUT PORT HERE"
```

7. Paste the contents of the replica configuration file in here. Initiate the replica set

```
> rs.initiate(cfg)
{
        "info" : "Config now saved locally.  Should come online in about a minute.",
        "ok" :
}
```

8. Refresh the configuration until you see PRIMARY and SECONDARY replica set members. Below is an example of what it looks like when working.

```
> rs.status()
> rs.status()
> rs.status()
replica0:PRIMARY> rs.status()
{
        "set" : "replica0",
        "date" : ISODate("2014-06-05T13:19:09Z"),
        "myState" : 1,
        "members" : [
                {
                        "_id" : 0,
                        "name" : "192.168.1.1:27017",
                        "health" : 1,
                        "state" : 1,
                        "stateStr" : "PRIMARY",
                        "uptime" : 255,
                        "optime" : Timestamp(1401974323, 1),
                        "optimeDate" : ISODate("2014-06-05T13:18:43Z"),
                        "electionTime" : Timestamp(1401974336, 1),
                        "electionDate" : ISODate("2014-06-05T13:18:56Z"),
                        "self" : true
                },
                {
                        "_id" : 1,
                        "name" : "192.168.1.2:27017",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 25,
                        "optime" : Timestamp(1401974323, 1),
                        "optimeDate" : ISODate("2014-06-05T13:18:43Z"),
```

```
                        "lastHeartbeat" : ISODate("2014-06-05T13:19:08Z"),
                        "lastHeartbeatRecv" : ISODate("2014-06-05T13:19:08Z"),
                        "pingMs" : 0,
                        "syncingTo" : "192.168.1.1:27017"
                },
                {
                        "_id" : 2,
                        "name" : "192.168.1.3:27017",
                        "health" : 0,
                        "state" : 8,
                        "stateStr" : "SECONDARY",
                        "uptime" : 0,
                        "optime" : Timestamp(0, 0),
                        "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
                        "lastHeartbeat" : ISODate("2014-06-05T13:19:08Z"),
                        "lastHeartbeatRecv" : ISODate("1970-01-01T00:00:00Z"),
                        "pingMs" : 0
                        "syncingTo" : "192.168.1.1:27017"
                }
        ],
        "ok" : 1
}
```

9. Explore the replica set a bit more.

```
docker logs db-1
docker logs db-1
docker logs db-3
gear status db-1
systemctl status ctr-db-2
```

10. Clean up the environment

```
gear list-units
docker ps
gear delete db-{1,2,3}
gear list-units
docker ps
```

7.5. Multi-host Application Linking - SKIP THIS SECTION, IT NEEDS WORK

On host 1:

1. Check the environment

```
gear list-units
docker ps
```

2. Explore the *http_single.json* file

```
cat http_single.json
{
  "Containers": [
    {
      "Name": "web-server",
```

```
      "Image": "demo/apache",
      "PublicPorts": [
        {
          "Internal": 80
        }
      ],
      "Links": [
        {
          "To": "web-server",
          "NonLocal": true,
          "MatchPort": true
        }
      ],
      "Count": 2
    }
  ],
  "IdPrefix": "",
  "RandomizeIds": false
}
```

3. Ensure that the geard and Docker daemons are running on the second host.

```
systemctl status docker  # on host 1
systemctl status docker  # on host 2
systemctl status geard  # on host 1
systemctl status geard  # on host 2
```

4. Deploy the application on both hosts, where **x.x.x.x** is the IP address of the second host

```
gear deploy http_single.json localhost x.x.x.x
```

5. List the units and containers on both hosts

```
gear list-units    # on host 1
gear list-units    # on host 2
docker ps          # on host 1
docker ps          # on host 2
```

6. On host 1, get the pid for the web server container

```
docker inspect --format '{{ .State.Pid }}' <container uid>
```

7. Use *nsenter* to enter the namespace of the PID and take a look at the IPtables rules. You will see that there is a rule forwarding all traffic to *192.168.1.x* to the external port on the localhost and the external port on the remote host. Basically geard is telling the container that every application is local.

```
nsenter -m -u -n -i -p -t <PID FROM <container uid>> bash
iptables -nvL -t nat
```

8. Ensure that you can get the index.html from each host

```
curl http://localhost:<external port localhost>
curl http://localhost:<external port remote host>
```

9. On host 2, ensure that you can pull that web page as well and compare to the output that you got inside the container on host 1

```
docker ps
```

```
curl http://localhost:<external port localhost>
```

10. Clean up the environment

```
gear list-units   # on host 1
gear list-units   # on host 2
gear delete web-server-1    # on host 1
gear delete web-server-2    # on host 2
```

7.6. SSH Enablement for Containers

On host 1:

TBD

Chapter 8. Upgrading and Rolling back a Project Atomic Host

8.1. Project Atomic Host Upgrade and Rollback

This section needs to be filled out a bit more and explain what's going on behind the scenes.

1. Read the man page.

```
man rpm-ostree
```

2. Check help

```
rpm-ostree --help
```

3. Check out the entries in */boot*. Notice that *loader* is a symlink. That's how the atomic updates work, by swaping that symlink after an *rpm-ostree upgrade*

```
cd /boot
ls -l
cat loader/entries/ostree-rh-atomic-controller-0.conf
cat loader/entries/ostree-rh-atomic-controller-1.conf
```

4. Notice one of the entries in the ostree-rh-atomic-controller-0.conf. The UUID will be different on your host. But you need to explore that directory structure as that is where you are booting into.

```
ostree=/ostree/boot.1/rh-atomic-controller/426d63397efa20a7eb72ae55e2243d11b37dc4cfab4
```

5. Explore the */ostree* directory.

```
cd /ostree; ls
cd /ostree/boot.1/rh-atomic-controller/ostree/boot.1/rh-atomic-controller/<YOUR UUID>/
```

6. Rollback to the prior install on Atomic Controller 1 by using the *rpm-ostree rollback* command. First, grab a copy of the RPMs installed, then compare to after the rollback.

```
rpm -qa | sort > /root/before-rollback

rpm-ostree rollback

Moving 'aab68bc5a9c24b08ffa2eb10e9c7ca4c572258fa9a5ef1e34d3c12e60920f389.0' to be firs
Transaction complete; bootconfig swap: yes deployment count change: 0)
Changed:
  cockpit-0.8-1.el7.x86_64
<snip>
  subscription-manager-1.10.14-7.el7.x86_64
Removed:
  c-ares-1.10.0-3.el7.x86_64
  cups-libs-1:1.6.3-14.el7.x86_64
  sssd-ldap-1.11.2-65.el7.x86_64
<snip>
  sssd-proxy-1.11.2-65.el7.x86_64
```

```
  subscription-manager-plugin-ostree-1.11.7-1.git.95.3f56593.el7.x86_64
  tar-2:1.26-29.el7.x86_64
Sucessfully reset deployment order; run "systemctl reboot" to start a reboot
```

7. Reboot to go back to the original install

```
reboot
```

8. Now compare the RPMs that are installed.

```
rpm -qa | sort | diff /root/before-rollback -
```