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 and Docker are very fast moving spaces at the moment. There is a already lot of existing documentation out there. This lab attempts to make use of some of that documentation and provide a comprehensive example of an end to end deployment using Project Atomic components. Considering how fast moving this space is, if you use this guide and find some steps, or anything out of date, please submite a PR and let’s get it fixed. I will try to include links to external documentation that I borrowed from when possible.

Chapter 2. Configure Project Atomic VMs

2.1. Options for Deployment

There are a few different ways to obtain Atomic images.

1. In .qcow2 format
2. Deploy from rpm

2.2. Obtaining the images

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

1. Create the directory structure.

```
cd ~
mkdir -p atomic/{1,2}
```

2. Download and decompress the images to the appropriate directories

```
cd ~
for i in 1 2; do
  pushd atomic/$i/;
  wget http://rcm-img06.build.bos.redhat.com/images/auto/rh-atomic-controller-el7-x86_
  xz -d latest-qcow2.xz;
  popd;
done
```

3. Install the image and create an additional drive

```
cd ~
for i in 1 2; do
  pushd atomic/$i/;
  sudo virt-install \
  --memory 2048 \
  --name Atomic_Host_$i \
  --disk latest-qcow2 \
  --import \
  --noautoconsole \
  --disk /home/scollier/atomic/$i/storage_$i.qcow2,format=qcow2,sparse=true,size=10;
  popd;
done
```

4. Get IP Address of both hosts and open up a terminal console to each machine.
5. Partition the extra drive, on both hosts.

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.

```
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
```

2.3. Validate Storage

Atomic images in the .qcow2 format do not have a lot of storage for Docker images. You will need to add an extra disk for */var/lib/docker*.

2.4. Validate Networking

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

```
hostnamectl     # on host 1
hostnamectl     # on host 2
```

2.5. Validate SELinux

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

```
getenforce
```

2.6. Ensure Docker and geard are Functional

+

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

Chapter 3. Building Docker Images

3.1. Building Docker Images

Need to put instructions for Building docker images

Chapter 4. Docker Basics

4.1. Docker

This lab has 3 sections

1. Overview
2. Lab 1: Docker Environment
3. Lab 2: Containers can Talk

4.2. Overview of Docker on Red Hat Enterprise Linux lab

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 to be 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.

4.3. Assumptions

This lab manual assumes that you are attending an instructor-led training class and that you will be using this lab manual in conjunction with the lecture.

This manual also assumes that you have been granted access to a single Red Hat Enterprise Linux server with which to perform the exercises.

A working knowledge of Linux-based tools and services such as telnet, Apache, etc… are assumed. If you do not have an understanding of any of these technologies, please let the instructors know.

4.4. Lab Environment

Red Hat Enterprise Linux provides shared services for Docker. A couple of these shared services are systemd and selinux. This lab will help to familiarize you with the common actions performed on Docker containers and images. The first part of the lab starts out on the host machine, the machine that runs all the containers. Then we'll move on to container inspection.

Accessing the Environment: You will have a virtual machine running on this host. You'll need to SSH into that virtual machine to complete the labs. You can open *virt-manager* to get the IP Address, then SSH into it from the workstation.

1. SSH Access

```
ssh root@10.16.143.125
```

4.5. 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

4.6. Lab 1: Docker Overview

4.7. Explore Docker

All actions in this lab will performed by the user *root*.

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-io
man docker
man docker-run
```

```
docker --help
docker run --help
```

3. A Docker *image* is basically a layer. A layer never changes. To take a look a the images that are on this system.

```
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 rhel7 echo hello
```

5. You won't see any return value. Where did it go? 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>
docker run -t rhel7 echo hello
```

6. 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
```

7. Check out the IP address of the container and also look at the route. You can see that the default route is that of the *docker0* bridge on the host.

```
ip a
ip r s
```

8. 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.

```
hostname
```

9. What processes are running inside the container?

```
ps aux
```

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

```
ps -Z
```

4.8. Saving Content

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 rhel7 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
```

4.9. 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.

```
brctl show
```

4. Check out the bridge and 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 s docker0
```

5. 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
```

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

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

7. 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
cd /var/lib/docker/devicemapper/mnt/<Container ID><tab><tab>/rootfs
```

8. 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>
```

9. 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>
```

10. 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
```

4.10. 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.

```
file /dev/log
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 from Summit"
exit
```

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

```
journalctl | grep -i "This is a log from Summit"
```

4.11. 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 *summit/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.

```
[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
```

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
```

4.12. 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."

Note

All images have been built before labtime. If you would like to review what was used, all Dockerfiles are in */root/summit_link_demo*.

4.13. 2.1 MariaDB

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

4.14. 2.1.1 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/summit_link_demo/mariadb; ls
```

2. Review the Dockerfile. Look at the *Dockerfile*. From the contents below, you can see that the Dockerfile is starting with the RHEL7 base image and is maintained by Stephen Tweedie. 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. The remaining commands are *ADD*, which are used to add content to the image and 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.

```
# cat Dockerfile
FROM fedora:20
MAINTAINER Stephen Tweedie <sct@redhat.com>

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

VOLUME [ "/var/lib/mysql" ]

ADD ./start.sh /start.sh
ADD ./supervisord.conf /etc/supervisord.conf

RUN chmod 755 /start.sh

EXPOSE 3306

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

3. Review the supervisord.conf file

Straight from the supervisord.org site:

"Supervisor: A Process Control System

Supervisor is a client/server system that allows its users to monitor and control a number of processes on UNIX-like operating systems."

There are a couple of reasons to use *supervisord* inside a container. The first is that Docker really only wants to be in charge of one service. So if you are running multiple services in a POC container such as MariaDB and Apache at the same time, you need a way to manage those. Present *supervisord* as the service that runs on launch and let it control the other services in the background. Also, supervisord can run services in foreground mode. Docker likes that.

The *supervisord.conf* file instructs the *supervisord* daemon as to which processes it is responsible for. This *supervisord.conf* file has been pared down considerably.

+

```
# cat supervisord.conf
[unix_http_server]
file=/tmp/supervisor.sock   ; (the path to the socket file)

[supervisord]
logfile=/tmp/supervisord.log ; (main log file;default $CWD/supervisord.log)
logfile_maxbytes=50MB        ; (max main logfile bytes b4 rotation;default 50MB)
logfile_backups=10           ; (num of main logfile rotation backups;default 10)
loglevel=info                ; (log level;default info; others: debug,warn,trace)
pidfile=/tmp/supervisord.pid ; (supervisord pidfile;default supervisord.pid)
nodaemon=false               ; (start in foreground if true;default false)
minfds=1024                  ; (min. avail startup file descriptors;default 1024)
minprocs=200                 ; (min. avail process descriptors;default 200)
```

```
[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface

[supervisorctl]
serverurl=unix:///tmp/supervisor.sock ; use a unix:// URL  for a unix socket

[program:mariadb]
command=/usr/bin/mysqld_safe
stdout_logfile=/var/log/supervisor/%(program_name)s.log
stderr_logfile=/var/log/supervisor/%(program_name)s.log
autorestart=true
```

1. Review the start.sh script The *start.sh* script is called by the container to start the *supervisord* daemon. The first thing the *start.sh* script does is checks to see if the database has been created yet. If it has, just start the container, if not, create it. The reason for this is this container uses a shared volume. It only needs to create the database one time. All other times the container starts, use existing data.

```
# cat start.sh
#!/bin/bash -x

__mysql_config() {

if [ ! -f /mariadb/db/ibdata1 ]; then
  echo
  echo "Database does not exist, creating now."
  echo
  sleep 2
  mysql_install_db
  chown -R mysql:mysql /var/lib/mysql
  /usr/bin/mysqld_safe &
  sleep 10

  echo "Running the start_mysql function."
  mysqladmin -u root password mysqlPassword
  mysql -uroot -pmysqlPassword -e "CREATE DATABASE testdb"

  mysql -uroot -pmysqlPassword -e "GRANT ALL PRIVILEGES ON testdb.* \
  TO 'testdb'@'localhost' IDENTIFIED BY 'mysqlPassword'; FLUSH PRIVILEGES;"

  mysql -uroot -pmysqlPassword -e "GRANT ALL PRIVILEGES ON *.* \
  TO 'testdb'@'%' IDENTIFIED BY 'mysqlPassword' WITH GRANT OPTION; FLUSH PRIVILEGES;"

  mysql -uroot -pmysqlPassword -e "GRANT ALL PRIVILEGES ON *.* \
  TO 'root'@'%' IDENTIFIED BY 'mysqlPassword' WITH GRANT OPTION; FLUSH PRIVILEGES;"

  mysql -uroot -pmysqlPassword -e "select user, host FROM mysql.user;"
  killall mysqld
  sleep 10
fi
}

__run_supervisor() {
echo "Running the run_supervisor function."
supervisord -n
}

# Call all functions
__mysql_config
```

```
__run_supervisor
```

4.15. 2.1.1 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 -d -v /mariadb/db:/var/lib/mysql -p 3306:3306 --name mariadb summit/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.

   ```
   ls -lZd /mariadb/db
   chcon -Rvt svirt_sandbox_file_t /mariadb/db/
   ```

6. Now launch the container again. First the container will have to be removed because of a naming conflict.

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

7. Launch the container again.

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

The container should be running at this time.

4.16. 2.2 Mediawiki

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

4.17. 2.2.1 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 to save time. 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/summit_link_demo*

1. Review the Dockerfile

```
cat Dockerfile
FROM scollier/apache
MAINTAINER Stephen Tweedie <sct@redhat.com>

# Basic RPM install...
RUN yum -y update; yum clean all

# Install:
#  Mediawiki, obviously
#  php, because mediawiki doesn't by itself install php into apache
#  php-mysqlnd: this image will be configured to run against the
#               Fedora-Dockerfiles mariadb image so we need the mysqld
#               client support for php
RUN yum -y install mediawiki php php-mysqlnd; yum clean all

# 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 ./run-apache.sh /run-apache.sh
ADD ./LocalSettings.php /var/www/html/wiki/
RUN chmod +x /run-apache.sh
RUN chmod +x /config.sh
RUN /config.sh

# localhost:/wiki/mw-config should now be available to configure mediawiki.

# Add script to update the IP address of a linked mariadb container if
# needed:

ADD run-mw.sh /run-mw.sh
RUN chmod +x /run-mw.sh
CMD ["/run-mw.sh"]
```

2. Review the config.sh script

```
# cat config.sh
#!/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

```
# cat run-mw.sh
#!/bin/bash
#
# Run mediawiki in a docker container environment.

function edit_in_place () {
    tmp=`mktemp`
    sed -e "$2" < "$1" > $tmp
    cat $tmp > "$1"
    rm $tmp
}

# 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
    edit_in_place /usr/share/mediawiki/includes/DefaultSettings.php 's/^\$wgDBserver =

    # 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
        edit_in_place /var/www/html/wiki/LocalSettings.php 's/^\$wgDBserver =.*$/\$wgD
        sed -i 's/^\$wgServer =.*$/\$wgServer = "http:\/\/'$HOST_IP'";/' /var/www/html
    fi
fi


# Finally fall through to the apache startup script that the apache
# Dockerfile (which we build on top of here) sets up
exec /run-apache.sh
```

4.18. 2.2.2 Launch the Mediawiki Container

This section show's 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 virtual machine.

+

```
ip a

docker run -d -e=HOST_IP=IP_OF_VIRTUAL_MACHINE --link mariadb:db  -v /var/www/html/ -p 8
```

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:

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

2. Take a look at the logs for the container and notice how the IP substitutions were done. One IP address is for the MariaDB host and one IP address is the virtual machine IP address. It's the same IP address that was passed via the *docker run* command.

```
docker logs mediawiki
```

3. Open browser on the host running the VM and confirm the configuration is complte.

```
firefox &
```

4. 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
```

5. 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<br>
Passwrod: redhat
```

6. 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
    edit_in_place /usr/share/mediawiki/includes/DefaultSettings.php 's/^\$wgDBserver =

    # 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
        edit_in_place /var/www/html/wiki/LocalSettings.php 's/^\$wgDBserver =.*$/\$wgD
        sed -i 's/^\$wgServer =.*$/\$wgServer = "http:\/\/'$HOST_IP'";/' /var/www/html
    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.

4.19. Continue your Learning

4.20. 3.1 How to Install

On a Fedora host

```
yum install fedora-dockerfiles docker-io
```

4.21. 3.2 More Information

Project Atomic site:

http://projectatomic.io

Chapter 5. Introduction to geard

5.1. 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

5.2. geard Lab Prerequisites

1. Two Atomic hosts

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

2. 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
```

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

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

5.3. Part 1. Deploy a Single Container

On 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/apache web-server --start -p 80:80
```

4. List the units again

```
gear list-units
```

5. Show the container is also recognized by Docker

```
docker ps
```

6. Make sure the web server is responding to requests

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

7. List the units one more time

```
gear list-units
```

8. Clean up the environmentnd

```
gear delete web-server
gear list-units
```

5.4. Part 2. Deploy Multiple Containers on a Single Host

On host 1:

1. Check the geard environment

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

2. Check out the *http_single_HTB.json* file

```
cat http_single_HTB.json
{
  "Containers": [
    {
      "Name": "web-server",
      "Image": "demo/apache",
      "PublicPorts": [
        {
          "Internal": 80
        }
      ],
      "Links": [
        {
          "To": "web-server",
```

```
          "NonLocal": true,
          "MatchPort": true
        }
      ],
      "Count": 3
    }
  ],
  "IdPrefix": "",
  "RandomizeIds": false
}
```

This file tells geard to deploy an application with *Name* "web-server", to use *image* "demo/apache" with an internal port of 80 and a count of 3.

1. Check out the */var/lib/containers/* directory. This is where geard stores it's content. Right now there will not be any content in the *units* directory. After a successful launch of the application there will be.

```
ls /var/lib/containers/
ls /var/lib/containers/units/
```

2. Deploy the application. This tells geard to deploy all three web servers on the same host.

```
gear deploy http_single_HTB.json
```

3. Check the */var/lib/containers/* directory again to see what content geard added

```
ls /var/lib/containers/
ls /var/lib/containers/units/
```

4. Check *systemd* integration

```
systemctl status ctr-web-server-1
```

5. Check to ensure the web servers are registered with geard and Docker

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

6. Now that we have the ports listed as shown in *docker ps*, let's make sure one of the web servers are running.

```
curl http://localhost:4000
```

7. Clean up the environment

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

5.5. Part 3. 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.

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

3. Deploy the application

```
gear deploy mongo_deploy.json
```

4. List the units and container

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

5. Connect with the MongoDB client

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

6. Copy in the configuration file

```
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"
        },
    ]
}
```

7. Initiate the replica set

```
> rs.initiate(cfg)
```

8. Refresh the configuration until you see PRIMARY and SECONDARY

```
> rs.status()
> rs.status()
> rs.status()
```

9. Clean up the environment

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

5.6. Part 4. Multi-host Application Linking

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
```

5.7. Part 5. SSH Enablement for Containers

On host 1:

TBD

Chapter 6. Upgrading and Rolling back a Project Atomic Host

6.1. Project Atomic Host Upgrade and Rollback

Need to put instructions for pulling a new tree and rollback