Contious Integration with Docker

Michael Klöckner

2014-01-29

Contents

1 Overview

1.1 Abstract

One major issue in continuous integration is to insure a once deployed build artifact never changes in future deployment scenarios. Many development cycles require short boot times and quick deployment ways. The virtualization tool *Docker* relies on the *LXC* container technologie. It stores containers or images in a repository, making them immutable afterwards. Pulling a particular image out of a repository and starting it as a container, results in starting the same build artifact in all future deployment scenarios.

As proposed by Markus Fix **we build immutable** *Docker* **images from** *Scala* **WebApp release artifacts and push them into a private** *Docker* **registry. We start the** *Scala* **WebApp service inside a** *Docker* **container pulled out of the private** *Docker* **registry.**

1.2 Layout

Chapter two shows different ways to install *Docker* and its prerequisites. It then gives a short introduction into basic *Docker* commands. Chapter three deals with the private registry. It condenses the main work we did. Chapter four and five combine a sample *Scala* WebApp with a *Jenkins* server. The next chapter

provides our experience with running the services on Amazon Web Service. We add our conclusions and open questions as extra chapters.

1.3 Author

- Author: Michael Klöckner, Weberstr. 39, 60318 Frankfurt am Main,
- Email: mkl AT im7 DOT de
- Phone: +49 69 9866 1103

1.4 *Docker* version

We installed *Docker* version 0.8 in 2014/02.

2 *Docker*

This chapter deals with installation issues and some basic *Docker* commands. It is mainly take from the Docker documentation.

2.1 Installing Docker

2.1.1 Kernel

Docker needs a 64-Bit Linux distribution, a recent kernel > 3.8 and LXC installed. Either you use a system with the appropriate kernel installed, or you update the kernel by hand as described in kernel compilation. The kernel needs to have compiled all options concerning virtual NICs, especially BRIDGED NICs, all NAT options and all net (NF) options. Download the kernel source, untar it, change into the directory and configure it properly. To compile the kernel as a debian package named **fora-kernel-3.13.3** to be installed later together with it's header follow these instructions:

```
make-kpkg clean
make-kpkg --append-to-version "-flora-kernel-3.13.3" --revision "1" \
--initrd kernel_image kernel_headers
```

The package is to be found one directory upwards and can be installed using

```
dpkg -i ../linux-headers-3.13.3-flora-kernel-3.13.3_1.2_amd64.deb \
../linux-image-3.13.3-flora-kernel-3.13.3_1.2_amd64.deb/.
# reboot
sudo reboot
```

We could also install a backport kernel:

```
# install the backported kernel
sudo apt-get update
sudo apt-get install  linux-image-generic-lts-raring  linux-headers-generic-lts-raring
# reboot
sudo reboot
```

2.1.2 Installing *Docker* by hand

First add the *Docker* repository key to your local keychain.

```
sudo apt-key adv --keyserver keyserver.ubuntu.com \
--recv-keys 36A1D7869245C8950F966E92D8576A8BA88D21E9
```

On AWS-machines we might be forced to pass protocol and port to the *gpg* utility explicitly:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 \
--recv-keys 36A1D7869245C8950F966E92D8576A8BA88D21E9
```

Add the *Docker* repository to your apt sources list, update and install the lxc-docker package.

```
sudo sh -c "echo deb http://get.docker.io/ubuntu docker main\
> /etc/apt/sources.list.d/docker.list"
sudo apt-get update
sudo apt-get install lxc-docker
```

Now verify that the installation has worked by downloading the ubuntu image and launching a container. `sudo docker run -i -t ubuntu /bin/bash`. Type exit to exit.

2.1.3 Installing *Docker* by script

Docker.io provides an installation script to be called: `curl -s https://get.docker.io/ubuntu/ | sudo sh` Now verify that the installation has worked by downloading the ubuntu image and launching a container. `sudo docker run -i -t ubuntu /bin/bash` Type exit to exit.

2.1.4 Installing *Docker* on AWS

Docker.io provides an installation guide for Amazon Web Services EC2.

- Choose an image:
  - Launch the Create Instance Wizard menu on your AWS Console.
  - Click the Select button for a 64Bit Ubuntu image. For example: Ubuntu Server 12.04.3 LTS.

  - For testing you can use the default (possibly free) t1.micro instance (more info on pricing).
  - Click the Next: Configure Instance Details button at the bottom right.

- Tell CloudInit to install *Docker*:
  - When you're on the Configure Instance Details step, expand the Advanced Details section.
  - Under User data, select As text
  - Enter #include https://get.docker.io into the instance User Data. CloudInit is part of the Ubuntu image you chose; it will bootstrap *Docker* by running the shell script located at this URL.

- After a few more standard choices where defaults are probably OK, your AWS Ubuntu instance with *Docker* should be running!

If this is your first AWS instance, you may need to set up your Security Group to allow SSH. By default all incoming ports to your new instance will be blocked by the AWS Security Group, so you might just get timeouts when you try to connect. Installing with get.docker.io (as above) will create a service named *lxc-docker*. It will also set up a *Docker* group and you may want to add the ubuntu user to it so that you don't have to use sudo for every *Docker* command.

We found, that launching an AWS instance of type Ubuntu Server 12.04 LTS (PV) - ami-fa9cf1ca (64-bit) does not come with a 3.8 kernel and CloudInit did not install *Docker*, so we had to install both manually.

2.1.5 Configuration

- The daemon's config file is placed in *etc/default/Docker*.
- Images, containers and their configurations are placed under *var/lib/Docker*.

2.2 Play with *Docker*

We describe some basic *Docker* commands.

2.2.1 Check your *Docker* installation.

```
# Check that you have a working install
docker info
```

2.2.2 Download a pre-built image

```
# Download an ubuntu image
sudo docker pull ubuntu
```

2.2.3 Run an interactive shell

```
# Run an interactive shell in the ubuntu image,
# allocate a tty, attach stdin and stdout
# To detach the tty without exiting the shell,
# use the escape sequence Ctrl-p + Ctrl-q
sudo docker run -i -t ubuntu /bin/bash
```

2.2.4 Bind to a port

The *Docker* client can use -H to connect to a custom port. -H accepts host and port assignment in the following format:

- tcp://[host][:port] =
- unix://path =
- host[:port] or :port =

```
# Run docker in daemon mode
sudo <path to>/docker -H 0.0.0.0:5555 -d &
# Download an ubuntu image
sudo docker -H :5555 pull ubuntu
```

2.2.5 Starting a long run

```
# Start a very useful long-running process
JOB=$(sudo docker run -d ubuntu /bin/sh -c "while true; \
do echo Hello world; sleep 1; done")
# Collect the output of the job so far
sudo docker logs $JOB
# Kill the job
sudo docker kill $JOB
```

2.2.6 Bind a service on a TCP port

```
# Bind port 4444 of this container, and tell netcat to listen on it
JOB=$(sudo docker run -d -p 4444 ubuntu:12.10 /bin/nc -l 4444)

# Which public port is NATed to my container?
PORT=$(sudo docker port $JOB 4444 | awk -F: '{ print $2 }')

# Connect to the public port
echo hello world | nc 127.0.0.1 $PORT

# Verify that the network connection worked
echo "Daemon received: $(sudo docker logs $JOB)"
```

2.2.7 Committing (saving) a container state

Save your containers state to a container image, so the state can be re-used. When you commit your container only the differences between the image the container was created from and the current state of the container will be stored (as a diff). See which images you already have using the *Docker* images command.

```
# Commit your container to a new named image
sudo docker commit <container_id> <some_name>

# List your containers
sudo docker images
```

2.2.8 Committing a Container to a Named Image

When you make changes to an existing image, those changes get saved to a container's file system. You can then promote that container to become an image by making a commit. In addition to converting the container to an image, this is also your opportunity to name the image, specifically a name that includes your user name from the Central *Docker* Index (as you did a login above) and a meaningful name for the image.

```
# format is "sudo docker commit <container_id> <username>/<imagename>"
$ sudo docker commit $CONTAINER_ID myname/kickassapp
```

2.2.9 Pushing an image to its repository

In order to push an image to its repository you need to have committed your container to a named image (see above). Now you can commit this image to the repository designated by its name or tag.

```
# format is "docker push <username>/<repo_name>"
$ sudo docker push myname/kickassapp
```

2.2.10 Export a container

To export a container to a tar file just type:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
mkl/debian          7.4                 11ed3d47ec89        About an hour ago   117.8 MB
mkl/debian          latest              11ed3d47ec89        About an hour ago   117.8 MB
mkl/debian          wheezy              11ed3d47ec89        About an hour ago   117.8 MB
ubuntu              13.10               9f676bd305a4        2 weeks ago         182.1 MB
ubuntu              saucy               9f676bd305a4        2 weeks ago         182.1 MB

$ docker ps -a
```

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
ac3a595c294c        mkl/debian:7.4      /bin/bash           58 minutes ago      Exit 1
f7528d270208        mkl/debian:7.4      echo success        About an hour ago   Exit 0
6a569d77e974        ubuntu:12.04        /bin/bash           16 hours ago        Exit 0
```

```
$ docker export ac3a595c294c  > exampleimage.tar
```

2.2.11 Import a container

At this time, the URL must start with http and point to a single file archive (.tar, .tar.gz, .tgz, .bzip, .tar.xz, or .txz) containing a root filesystem. If you would like to import from a local directory or archive, you can use the - parameter to take the data from stdin. To import from a remote url type:

```
$ sudo docker import http://example.com/exampleimage.tar
```

To import from a local file type:

```
$ cat exampleimage.tar | sudo docker import - exampleimagelocal:new
```

Note the sudo in this example – you must preserve the ownership of the files (especially root ownership) during the archiving with tar. If you are not root (or the sudo command) when you tar, then the ownership might not get preserved.

2.2.12 Mount a volume

Docker provides the parameter `-v` with the `run` command to create a persistent storage device.

```
docker run -v /volume1  myName/debian true
```

runs the image `myName/debian` with command `true` and creates a volume attached to this container which is visible inside as `/volume1`. To mount the host directory `/opt/this-volume` to a container in read only mode, we prepend the host directory name to the volume name:

```
docker run -v /opt/this-volume:/volume1:ro  myName/debian true
```

If you remove containers that mount volumes, the volumes will not be deleted until there are no containers still referencing those volumes. This allows you to upgrade, or effectively migrate data volumes between containers. The complete syntax is

```
-v=[]: Create a bind mount with: [host-dir]:[container-dir]:[rw|ro].
```

If host-dir is missing from the command, then docker creates a new volume. If host-dir is present but points to a non-existent directory on the host, Docker will automatically create this directory and use it as the source of the bind-mount. Note that this is not available from a Dockerfile due the portability and sharing purpose of it. The host-dir volumes are entirely host-dependent and might not work on any other machine. Section Container and Images describes, where *Docker* stores the volumes mounted by the container.

2.3 Build your a base image

Docker.io provides a way to create a base image. The base image heavily depends on the distribution, the host is running. The example script mkimage-debootstrap.sh creates a debian base image.

2.3.1 Download the script

```
$ wget https://raw.github.com/dotcloud/Docker/master/contrib/mkimage-debootstrap.sh
$ chmod +x mkimage-debootstrap.sh
```

This downloads the build-script for a debian *Docker* base image.

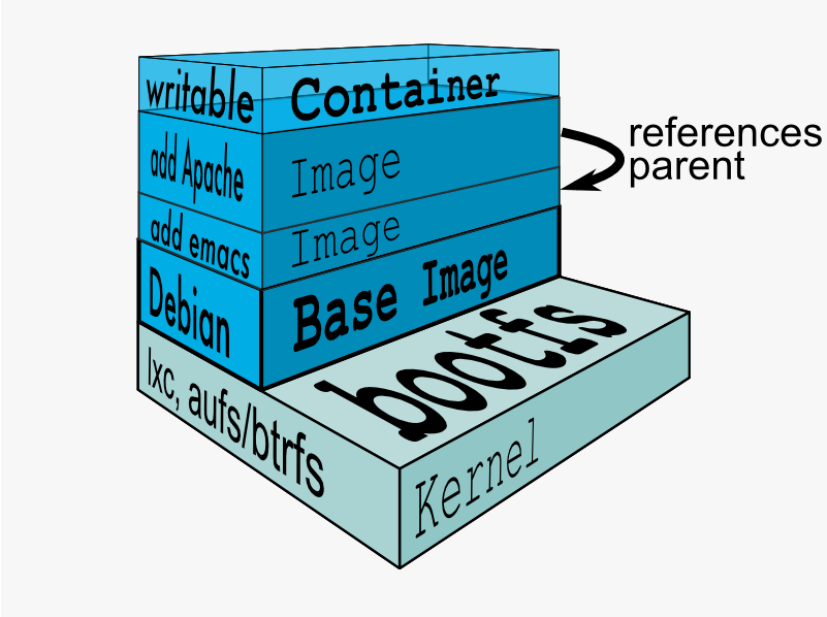
2.3.2 Build the base image

```
$ ./mkimage-debootstrap.sh flora/debian wheezy
$ docker images -a
```

This creates a new *Docker* base image for debain wheezy and puts it into ropsitory *flora/debian*, where *flora* is the username and *debian* the repo name.

2.4 Layers

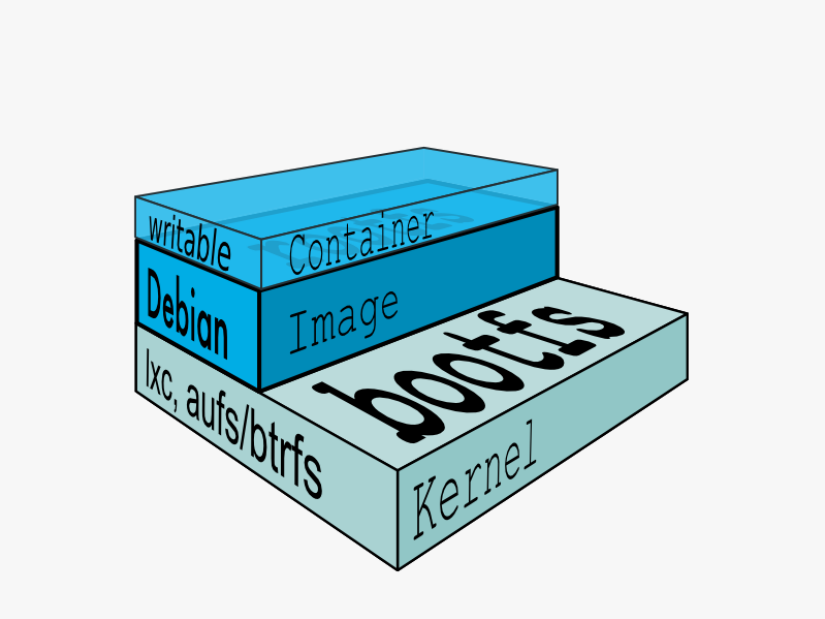
When Docker mounts the rootfs, it starts read-only, as in a traditional Linux boot, but then, instead of changing the file system to read-write mode, it takes advantage of a union mount to add a *read-write file system over the read-only file system*. In fact there may be multiple read-only file systems stacked on top of each other. We think of each one of these file systems as a **layer**.



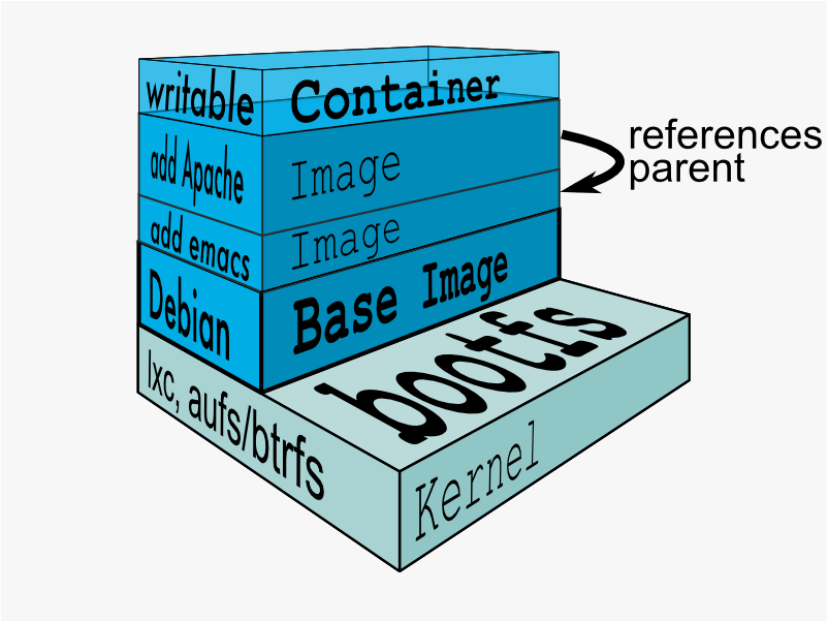
2.4.1 Union file system

At first, the top read-write layer has nothing in it, but any time a process creates a file, this happens in the top layer. And if something needs to update an existing file in a lower layer, then the file gets copied to the upper layer and changes go into the copy. The version of the file on the lower layer cannot be seen by the applications anymore, but it is there, unchanged. We call the union of the read-write layer and all the read-only layers a **union file system**.

2.4.2 Base Image



In Docker terminology, a read-only Layer is called an image. An image never changes.



Each image may depend on one more image which forms the layer beneath it. We sometimes say that the lower image is the parent of the upper image. An image that has **no parent** is a **base image**. All images are identified by a 64 hexadecimal digit string (internally a 256bit value). To simplify their use, a short ID of the first 12 characters can be used on the command line. There is a small possibility of short id collisions, so the docker server will always return the long ID.

2.5 Container and Images

As *Docker* is under heavy development, the file system storing *Docker* related information changes rapidly. The main directory to look for *Docker* relevant bits and bytes is *var/lib/Docker*. In this section **GUID** is the full blown container id as given by `docker ps -a -no-trunc`.

2.5.1 LXC configuration

Using the Linux Container package http://linuxcontainers.org/, *Docker* configures each container partly by setting lxc options in *var/lib/Docker/container/GUID/config.lxc*.

2.5.2 Container Root File System

The corresponding root file system is stored in *var/lib/Docker/devicemapper/mnt/GUID/rootfs*. Here GUID is the full blown container id as given by `docker ps -a -no-trunc`

2.5.3 Container Volumes

If a container mounts a volume from inside the files on that volume are stored under *var/lib/Docker/vfs/dir/GUID*. Data stored under these volumes are persistent between container runs. There is a way to share these volumes between containers.

2.5.4 Removing a Container or an Image

To remove a container from a repository we list the containers and type:

```
docker ps -a
docker rm GUID
```

To remove an image from a repository we list the images and type:

```
docker images -a
docker rmi USER/REPO:TAG
```

Here `USER/REPO:TAG` referes to the user part, the repository part and the tag part of a special image. Note that command `docker images -a` my list the same GUID multiple times as the same image may be tagged differently. Removing an image beeing tagged multiple times only results in deleting the tag, keeping the other tagged version(s) in the repository.

3 Private Registry

Right now (version 0.6), private repositories are only possible by hosting your private registry.

3.1 Pushing to a private repo

To push or pull to a repository on your own registry, you must prefix the tag with the address of the registry's host, like this:

```
# Tag to create a repository with the full registry location.
# The location (e.g. localhost.localdomain:5000) becomes
# a permanent part of the repository name
sudo docker tag 0u812deadbeef localhost.localdomain:5000/repo_name
# Push the new repository to its home location on localhost
sudo docker push localhost.localdomain:5000/repo_name
```

The push command will fail, if no registry server answers locally on port 5000.

3.2 Building a private registry

There are two ways to run a registry server: as a container or as gunicorn application. Sam Alba, dotCloud's first engineer hire, describes how to build the application, provided `gunicorn` and `pip` is installed:

```
#install pip
sudo apt-get install python-pip
#install gunicorn
sudo apt-get install gunicorn
#install the docker-registry container an run it
git clone https://github.com/dotcloud/docker-registry.git
cd docker-registry
cp config_sample.yml config.yml
sudo pip install -r requirements.txt
gunicorn --access-logfile - --log-level debug --debug \
    -b 0.0.0.0:5000 -w 1 wsgi:application
```

To simplify things, the github repository comes with a Dockerfile do build a container from Ubuntu 13.4. Once a repository has a registry's host name as part of the tag, we can push and pull it like any other repository. On the other hand our private registry will not be searchable (or indexed at all), and there will be no user name checking performed. Our registry will function completely independently from the Central Index at Docker.io.

3.3 Changes to the registry building code

The code posted by Sam Alba did not work out of the box neither on a Debian Wheezy (7.4) nor on an Ubuntu 12.4. We had to previously install and upgrade these packages on a docker host to get the *gunicorn* application or the registry container running:

3.3.1 Upgrade *pip*

```
wget https://raw.github.com/pypa/pip/master/contrib/get-pip.py -O get-pip.py
sudo python get-pip.py
```

3.3.2 Install *gcc*

```
sudo apt-get install -y gcc
```

3.3.3 Install deb-packages from file *docker-registry/Dockerfile*

```
sudo apt-get install -y  git-core build-essential python-dev \
    libevent-dev python-openssl liblzma-dev wget
```

3.4 Registry as a gunicorn application

```
git clone https://github.com/dotcloud/docker-registry.git
cd docker-registry
cp config_sample.yml config.yml
sudo pip install -r requirements.txt
gunicorn --access-logfile - --log-level debug --debug \
    -b 0.0.0.0:5000 -w 1 wsgi:application
```

Finallay the *gnuicorn* application worked as expected.

3.5 Registry as a container

An alternative way is to build a registry container after we installed the necessary libraries on the docker host.

```
git clone https://github.com/dotcloud/docker-registry.git
cd docker-registry
sudo docker build -rm -t registry .
sudo docker run -d -p 5000:5000 registry
```

This results in an image tagged *registry* and a container exposing port 5000 running on the same machine.

We provide a bash shell script to complete the task.

3.6 Registry as a Web Services

Starting from $12 per month for 5 repositories quay.io serves a private registry on the web. We did not find out, whether docker.io indexes quay.io hosted private repositories.

3.7 Testing the private registry

Using two machines, **registry.local** and **host01.local** both beeing known by local DNS, we build *Docker* images on **host01.local** and store them on **registry.local** running the registry on port 5000. Suppose, we just successfully built an image from a Dockerfile or committed a container -resulting in an image. Let's see, how to tag the image **foo**, push it into the registry, delete it locally and pull it out again later.

It is essential to know the exact tag for the image, as the private registry does not allow searching of tags or images.

3.7.1 Tag the image

```
# what have we got?
host01:$ sudo docker images
REPOSITORY                          TAG       IMAGE ID         CREATED           VIRTUAL SI
```

```
debian/foo                              foo       38332d781d61     2 minutes ago     699.4 MB
ubuntu                                  13.10     9f676bd305a4     3 weeks ago       182.1 MB
ubuntu                                  saucy     9f676bd305a4     3 weeks ago       182.1 MB

host01:$ sudo docker tag 38332d781d61 registry.local:5000/debian/foo
host01:$ sudo docker images
REPOSITORY                              TAG       IMAGE ID         CREATED           VIRTUAL SI
debian/foo                              foo       38332d781d61     2 minutes ago     699.4 MB
registry.local:5000/debian/foo          foo       38332d781d61     2 minutes ago     699.4 MB
ubuntu                                  13.10     9f676bd305a4     3 weeks ago       182.1 MB
ubuntu                                  saucy     9f676bd305a4     3 weeks ago       182.1 MB
```

3.7.2 Push the image into the registry

```
# push this image to the registry server
host01:$ sudo docker push registry.local:5000/debian/foo
The push refers to a repository [registry.local:5000/debian/foo] (len: 1)
Sending image list
Pushing repository registry.local:5000/debian/foo (1 tags)
11ed3d47ec89: Image successfully pushed
38332d781d61: Image successfully pushed
Pushing tag for rev [38332d781d61] on {http://registry.local:5000/v1/repositories/debian/foo
```

3.7.3 Remove the image locally

```
# remove the image locally by removing both tags referring to the same image!
host01:$ sudo docker rmi  debian/foo
Untagged:38332d781d616823aaaaadc7c9ca4243f696b4efe2a74a49eb18fd062633198d

host01:$ sudo docker rmi registry.local:5000/debian/foo
Untagged:38332d781d616823aaaaadc7c9ca4243f696b4efe2a74a49eb18fd062633198d

# check for local images
host01:$ sudo docker images
REPOSITORY                      TAG           IMAGE ID         CREATED           VIRTUAL SIZE
ubuntu                          13.10         9f676bd305a4     3 weeks ago       182.1 MB
ubuntu                          saucy         9f676bd305a4     3 weeks ago       182.1 MB
```

3.7.4 Pull the image out of the registry

```
# we pull the image  using the exact name we pushed it with:
host01:$ sudo docker pull registry.local:5000/debian/foo
Pulling repository host01.local:5000/debian/foo
38332d781d61: Download complete
9f676bd305a4: Download complete
```

```
#check the local images
host01:$ sudo docker images
REPOSITORY                          TAG       IMAGE ID         CREATED             VIRTUAL SIZ
registry.local:5000/debian/foo      foo       38332d781d61     2 minutes ago       699.4 MB
ubuntu                              13.10     9f676bd305a4     3 weeks ago         182.1 M
```

4 Installing a *Scala/Java* WebApp

As a proof of concept, we install a *Scala* WebApp with *Lift*. We need *Java* version > 6 and we use *Lift* as the framework.

4.1 Installing the necessary packages and *Java*

We need *jdk* at least version 6, *wegt*, *zip* and git:

```
$ apt-get update
$ apt-get install -y apt-utils
$ apt-get install -y openjdk-7-jre
$ apt-get install -y openjdk-7-jdk
$ apt-get install -y wget
$ apt-get install -y zip
$ apt-get install -y git
```

This installs Java 7 and my take a minute.

4.2 Installing *tomcat7*

We use *tomcat* as the **Apache Tomcat Servlet/JSP** engine to serve our *Scala* WebApp, installing it by typing:

```
$ apt-get update
$ apt-get install -y tomcat7
```

Tomcat serves servlets at http://localhost:8080. The debian package starts the service automatically at boot time via *etc/init.d/tomcat7* script.

4.3 Scala WebApp

4.3.1 Installation

We download and configure a sample *Scala* WebApp and unzip it under *opt*.

```
$ wget -O /tmp/master.zip https://github.com/Lift/Lift_26_sbt/archive/master.zip
$ unzip -d /opt/ /tmp/master.zip
```

4.3.2 Compiling the WebApp

The first time this process may take several minutes to download *maven* and the *Scala*-files. Later calls only compile the relevant jar- and war-files. To compile the WebApp we type:

```
$ cd /opt/lift_26_sbt-master/scala_210/lift_basic/ && ./sbt compile
```

/Lift/ web framework will download *sbt*, *Scala* and the necessary dependencies and compile the War-File **/opt/lift_26_sbt-master/scala_210/lift_basic/target/scala-2.10/lift-2.6-starter-template$_2$.10-0.0.3.war**.

By typing `/opt/lift_26_sbt-master/scala_210/lift_basic/sbt start` we should be able to see the WebApp at http://localhost:8080. To exit just type `exit`. The source of this WebApp is under **/opt/lift_26_sbt-master/scala_210/lift_basic/src/main/webapp/**. To prove the concept, we will later just change *index.html*.

4.3.3 Source Code Managemend

We created a public github repository to account for changes in the sample WebApp and trigger building *Docker* containers after a commit:

```
$ cd /opt/lift_26_sbt-master/scala_210/lift_basic/src
$ git init
$ git add *
$ git commit -m "INITIAL IMPORT"
$ git remote add origin https://github.com/radiomix/scala-blank.git
$ git push -u origin master
```

As the repo is not world writable, we provide the github user and password to push changes into the repo.

4.4 Deploying the WebApp to *tomcat7*

Lift uses *sbt* to compile the project and output a WAR- or JAR-file, which we want to copy into *tomcat7*'s webapp directory **/var/lib/tomcat7/webapps/**. We recompile the package and deploy it statically into *tomcat*.

```
$ cd /opt/lift_26_sbt-master/scala_210/lift_basic/
$  ./sbt  package
$ cp  target/scala-2.10/lift-2.6-starter-template_2.10-0.0.3.war \
    /var/lib/tomcat7/webapps/lift.war
$ service tomcat7 restart
```

This copies the war-file and restarts *tomcat7*. To see the WebApp direct your browser to http://localhost:8080/lift_basic/. There is no need to restart *tomcat* manually, as the *autoDeploy* attribute is set to "true" in file **/etc/tomcat7/server.xml**. *tomcat* even unpacks war-files if attribute *unpackWARs* is set to "true".

4.5 Building a container with the WebApp

We following Dockerfile will build an image containing the *Scala* WebApp war-file under *tomcat7*.

4.5.1 The Dockerfile

The command

```
sudo docker build -rm -t USER/REPO:TAG docker-dir/
```

builds the WebApp container using the docker file inside `docker-dir/` and pushes it into repository `USER/REPO` with tag `TAG`.

4.5.2 Multiple Dockerfiles

Creating directories for each build step, we can split building the image into different tasks. This eases testing of the `RUN` commands inside the docker files.

- `01\_openjdk7/Dockerfile` creates an image with *Java* and some utilities installed
- `02\_tomcat7/Dockerfile` installs *tomcat7* as the servlet engine
- `03\_install\_scala/Dockerfile` installs *Scala* and compiles a sample WebApp
- `04\deploy\_scala/Dockerfile` compiles the sample WebApp and copies the war-file into *tomcat7* webapp directory
- `scala/Dockerfile` installs the dependencies to host the *Scala* WebApp inside a *tomcat7* container in a *Docker* image

Note that each step in the installation process expects the previous image to be tagged properly. This can be avoided by concatenating the `RUN` commands from all the docker files into one single file as in the previous section.

4.5.3 Starting the container

The sample WebApp gets served by the *tomcat7* instance on port 8080. In order to expose this container port by the docker host we run the container typing:

```
sudo docker run -i -t -p :8080:8080 USER/REPO:TAG /bin/bash
```

5 Jenkins

This section describes how to install a *Jenkins* server, as described in https://wiki.jenkins-ci.org.

5.1 Installation

On Debian-based distributions, such as *Ubuntu*, you can install *Jenkins* through `apt-get`. Recent versions are available in an apt repository. Older but stable LTS versions are in this apt repository.

You need to have a JDK and JRE installed. openjdk-7-jre and openjdk-7-jdk are suggested. As root we type

```
wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key \
 | sudo apt-key add -
echo deb http://pkg.jenkins-ci.org/debian binary/ >  /etc/apt/sources.list.d/jenkins.list
apt-get update
apt-get install -y net-tools
apt-get install jenkins
```

What does this package do?

- *Jenkins* will be launched as a daemon up on start. See /etc/init.d/jenkins for more details.
- The 'jenkins' user is created to run this service.
- Log file will be placed in *var/log/jenkins/jenkins.log*. Check this file if you are troubleshooting Jenkins.
- *etc/default/jenkins will capture configuration parameters for the launch. - By default, /Jenkins* listen on port 8080. Access this port with your browser to start configuration.

5.2 Configure Jenkins

We want to run *Jenkins* on port 8090:

```
sed -i s/HTTP_PORT=8080/HTTP_PORT=8090/ /etc/default/jenkins
```

6 Deployment Scenario

Each time the WebApp has changed, it will be committed into a *git* repository. Via a web hook, *Jenkins* get's triggered to build a container with the committed source code. There are two ways to move the container around the net:

- a registry server provides containers on the net.
- an export/import mechanism stores containers as tar-files on the net.

Subsequently getting the base image can be done:

- out of the registry or
- imported as a tar file

Again moving a *Docker* container around the net can mean:

- push it into the registry and pull it out again later
- export it as a tar file and import this tar file later

6.1 Build and Tag the container

After pulling the source code from a *git* repo a Dockerfile is used to build the container with *tomcat7* and the *Scala* WebApp installed. Within this build process, the *Scala* WebApp gets compile and its jar-file copied into *tomcat7* webapp directory. *Docker* expexts the build process to take place in a context, which in our case is the directory containing the proper Dockerfile. We then type:

```
jenkins: $ sudo docker build -rm -t ubuntu/12-04:scala-ver-x .
```

Note the "." at the end of the line, indicating current directory! This builds a container and tags it `ubuntu/12-04:scala-ver-x`. It deletes the intermediate containers.

6.2 Deploying with a registry server

Suppose we use the registry server `registry.local` listening on port 5000.

6.2.1 Push the image to the registry

We tag the container properly by just rebuilding it resulting in just a second tag for the same container an then push the image into a registry.

```
jenkins: $ sudo docker build -rm -t registry.local:5000/ubuntu-12-04/scala-ver-x
jenkins: $ sudo docker push registry.local:5000/ubuntu-12-04/scala-ver-x
```

The second command pushes the image into the registry. We provide a Dockerfile file:image-builds/scala/Dockerfile to build the *Scala* WebApp from a fresh cloned git repository.

6.2.2 Move the container to production

On the production server we simply pull the newly build container:

```
prod: $ sudo docker pull registry.local:5000/ubuntu-12-04/scala-ver-x
```

6.3 Deploying with a tar file

Suppose we export and import containers to some storage on the net being accessible via http://storage.local.

6.3.1 Export the container to the storage device

We spot the container to be exported, obviously the last one after a successful build.

```
jenkins: $ sudo docker ps -a
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED          STATUS     POR
ac3a595c294c        ubuntu/scala:ver-x  /bin/bash           8 minutes ago    Exit 1
6a569d77e974        ubuntu:12.04        /bin/bash           16 hours ago     Exit 0

jenkins: $ sudo docker export  ac3a595c294c > scala-ver-x.tgz
jenkins: $ scp scala-ver-x.tgz http://storage.local/scala-ver-x.tgz
```

6.3.2 Move the container to production

We import the container from that storage device by typing

```
prod: $ sudo docker import http://my.storage.server.com/ubuntu-12-04/scala-ver-x.tgz
```

We could also split this into first loading the container tar file locally and then import it typing:

```
prod: $ sudo docker wget http://storage.local/ubuntu-12-04/scala-ver-x.tgz -O scala-ver-x.tg
prod: $ cat scala-ver-x.tgz | sudo docker import - ubuntu-12-04/scala-ver-x
```

There are multiple ways to save the container tar file locally.

6.4 Restart the new container

```
prod: sudo docker stop registry.local:5000/ubuntu-12-04/scala-ver-y
prod: sudo docker stop registry.local:5000/ubuntu-12-04/scala-ver-x
```

6.4.1 TODO provide a script for container export/import

6.4.2 TODO provide a script for container restart

7 Packer

We use *Packer* to build the *Docker*-capable AMI and the container running the *Scala* WebApp.

7.1 Installation

Following the steps in the *Packer* documentation we download the appropriate binary, unzip it and check the installation by typing `packer`.

7.2 AWS rights

Packer configures AMIs within json files. Source AMIs must be of virtualisation type 'paravirtual'. The AWS-user, whos credetials are used as parameters to run the build process, needs some IAM rights:

```
"Effect": "Allow",
      "Action" : [
        "ec2:AttachVolume",
        "ec2:CreateVolume",
        "ec2:DeleteVolume",
        "ec2:DescribeVolumes",
        "ec2:DetachVolume",

        "ec2:DescribeInstances",

        "ec2:CreateSnapshot",
        "ec2:DeleteSnapshot",
        "ec2:DescribeSnapshots",

        "ec2:DescribeImages",
        "ec2:RegisterImage",

        "ec2:CreateTags"
      ],
      "Resource" : "*"
```

7.3 Building a *Docker*-capable AMI

We used the following json file to build the *Docker*-capable AMI:

```
{
  "variables": {
    "aws_access_key": "",
    "aws_secret_key": "",
    "security_group_id": ""
  },
  "builders": [{
    "type": "amazon-ebs",
    "access_key": "{{user `aws_access_key`}}",
    "secret_key": "{{user `aws_secret_key`}}",
    "region": "us-west-2",
    "source_ami": "ami-fa9cf1ca",
    "instance_type": "t1.micro",
    "ssh_username": "ubuntu",
    "run_tags": {
        "Name": "docker-server-mkl-{{timestamp}}"
```

```
    },
    "ami_name": "docker-server-mkl-{{timestamp}}",
    "security_group_id": "{{user `security_group_id`}}"
  }],
  "provisioners": [{
    "type": "shell",
    "inline": [
      "sleep 30",
      "sudo apt-get update",
      "sudo apt-get install -y wget",
      "sudo apt-get install -y vi",
      "sudo apt-get install -y git",
      "sudo apt-get install -y mosh",
      "sudo apt-get install -y lynx",
      "sudo apt-get install -y unzip",
      "which lynx mosh git vi wget unzip",
      "sudo apt-get install -y linux-image-generic-lts-raring linux-headers-generic-lts-rari
      "sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 36A1D7869245C8
      "sudo sh -c 'echo deb http://get.docker.io/ubuntu docker main > /etc/apt/sources.list.
      "sudo reboot"
    ]
  },
 {
    "type": "shell",
    "inline": [
      "sleep 70",
      "echo INSTALLING PACKER",
      "wget https://dl.bintray.com/mitchellh/packer/0.5.2_linux_amd64.zip -O packer.zip",
      "sudo unzip -d /usr/local/bin packer.zip",
      "packer --version"
    ]
  },
 {
    "type": "shell",
    "inline": [
      "sleep 70",
      "sudo apt-get update",
      "echo INSTALLING DOCKER",
      "sudo apt-get install -y lxc-docker",
      "sudo docker --version",
      "sudo service ssh restart"
    ]
  }
 ]
}
```

Note the `variables` section, defining three variables that we do not want to store in the json file for security reasons. We use three provisioners of type shell to cope for the reboot after installing the appropriate kernel, install *Packer* and *Docker*. Suppose the above json file is called docker.json, we start the build process typing:

```
$ packer build -var 'aws_access_key=YOUR_AWS_ACCESS_KEY' \
    -var 'aws_secret_key=YOUR_AWS_SECRETE_KEY' \
    -var 'security_group_id=YOUR_SECURITY_GROUP_ID' \
    docker.json
```

This AMI is capable of running `docker` as well as `packer`, thus we will build *Docker* containers on this AMI.

7.4 Building a *Docker* container

Suppose the above *Packer* call provisioned an AMI with ID ami-f6e38dc6. This shall be the starting point to build *Docker* containers:

8 AWS

After testing the *Docker* installation locally we moved to Amazon Web Services launching four AMI of type Ubuntu Server 12.04 LTS (PV) - ami-fa9cf1ca (64-bit), attached an Elastic IP to each and run them as:

- a *Docker* registry server to keep the *Docker* images exposing port 5000
- a development server without *Docker* installed running *tomcat7* to develop the *Scala* WebApp on port 80
- a *Jenkins* server with *Docker* installed, to build the *Docker* images and push them to the registry server on port 5000
- a production server with only *Docker* installed, pulling the *Docker* images from the registry server and running the *Scala* WebApp within a container exposing port 80

We used a public Github account to version the *Scala* source code and trigger the *Jenkins* built.

This was our experience:

- The AMI we used (Ubuntu 12.04 LTS) does not come with a 3.8 kernel, we installed it manually.
- CloudInit did not install *Docker*, we installed it manually.
- Rebooting the registry AMI kept the *Docker* registry container running, both rebooting it within a shell as through the AWS web interface.

AWS Security Groups must take into account the ports *Docker* uses, espacially registry port 5000.

9 Conclusion

Docker is a robust and convenient virtualization tool that creates immutable images and speeds up software development and deployment essentially.

On our test machines as well as in an AWS cloud we installed a sample *Scala* WebApp, registered the source code with github and built a docker image each time the source code changed in *git*. We did not use *Jenkins* to build the image, but called a script by hand.

Building immutable images means running the build process for each release. This is cost and time intensive. Building just a container cuts time and costs effectively. The build process is not run as frequently as a machine reboot and may use resources an every day working AMI should not depend on.

We recommend splitting the build process into different parts:

- a version zero AMI providing *Docker* and necessary tools such as *git*, *vi*, *ssh*, . . .
- a *Docker* base container providing version zero of the application
- a *Docker* deployment container being rebuild each time a new release is deployed.

As a private registry server does not index the images, proper tagging is essential to pulling the proper container image. Once the image is pulled, it is indexed locally. But each time a machine boots on the cloud it depends on the registry server being alive. For each working environment development, staging and testing we would need a different registry server.

We therefor recommend to provide the AMIs with at least the base container and only fetch deltas to the deployment container at boot time. Though building such an AMI is expensive, this only takes place rarely.

An alternative way to load containers onto a *Docker* host are the `docker export` and `docker import` commands allowing for a convenient way to store images as a tar-file somewhere on the file system. We recommend storing the containers in an Amazon S3 bucket.

An application may provide it's configuration in a proprietary *git* repository, delivering the config files to be added during the build process, which then takes place within a fresh cloned version of this repo.

Using a tool like *Packer* would ease creating the different AMI needed.

- *Packer* creates a base AMI ready to run *Docker*, a *Docker* host.
- *Packer* builds a *Docker* base container running the application in version zero.
- *Packer* imports the base container into a *Docker* host and freezes this AMI.
- *Docker* host only gets the latest release of the application container -either from the registry or by importing it locally- and starts it.

Fetching the latest release of the application results in an immutable image although different deltas are pulled because each delta down to version zero of the application is itself an immutable image.

10 Open Questions

10.1 Networking

- Does the container get its own NIC or does it only expose ports?
- Which services are on what ports on the docker host as well as in the container?
- What is the standard maintenance port?
- What firewall rules are involved?

10.2 Logging

- How is logging done on the docker host as well as in the container?

10.3 Persistence

- What volumes are added to a container to store its data persistently?
- What happens with a write-processe if either the docker host or the container gets down? This could happen, if an AMI gets moved by AWS within an availability zone.

10.4 Cleaning Up

- As we start and stop containers, we have to keep track of those to be keept and those dispensable.
- The same holds true for images.
- We need a way to tag images properly for later searches.