

# README

Michael Klöckner

February 24, 2014

## Contents

<b>1</b>	<b>Overview</b>	<b>2</b>
1.1	Purpose . . . . .	2
1.2	Author . . . . .	3
1.3	<i>Docker</i> version . . . . .	3
1.4	Deployment workflow . . . . .	3
<b>2</b>	<b><i>Docker</i></b>	<b>4</b>
2.1	<b>TODO</b> explain <i>docker</i> terminology: Image, Container, Repository, Registry, Index. . . . .	4
2.2	Installation . . . . .	4
2.2.1	Kernel options . . . . .	4
2.2.2	Installation by hand . . . . .	4
2.2.3	Installation by script . . . . .	5
2.2.4	Installing on AWS . . . . .	5
2.2.5	Configuration . . . . .	6
2.3	Play with <i>Docker</i> . . . . .	6
2.3.1	Check your <i>Docker</i> installation. . . . .	6
2.3.2	Download a pre-built image . . . . .	6
2.3.3	Run an interactive shell . . . . .	6
2.3.4	Bind to a port . . . . .	6
2.3.5	Starting a long run . . . . .	7
2.3.6	Bind a service on a TCP port . . . . .	7
2.3.7	Committing (saving) a container state . . . . .	7
2.3.8	Committing a Container to a Named Image . . . . .	8
2.3.9	Pushing an image to its repository . . . . .	8
2.3.10	Private Repositories . . . . .	8
2.3.11	Export a container . . . . .	9

2.3.12	Import a container . . . . .	9
2.3.13	Mount a volume . . . . .	9
2.4	Build your own base image . . . . .	10
2.4.1	Download the script . . . . .	10
2.4.2	Build the base image . . . . .	10
2.5	Layers . . . . .	11
2.5.1	Union file system . . . . .	11
2.5.2	Base Image . . . . .	12
2.6	Container and Images . . . . .	13
2.6.1	LXC configuration . . . . .	13
2.6.2	Container Root File System . . . . .	13
2.6.3	Container Volumes . . . . .	13
2.6.4	Removing a Container or an Image . . . . .	13
<b>3</b>	<b>Installing a <i>Scala/Java</i> WebApp</b>	<b>14</b>
3.1	Installing the necessary packages and <i>Java</i> . . . . .	14
3.2	Installing <i>tomcat7</i> . . . . .	14
3.3	Scala WebApp . . . . .	14
3.3.1	Installation . . . . .	14
3.3.2	Compiling the WebApp . . . . .	15
3.4	Deploying the WebApp to <i>tomcat7</i> . . . . .	15
3.5	Building a container with the WebApp . . . . .	15
3.5.1	The Dockerfile . . . . .	15
3.5.2	Starting the container . . . . .	16
<b>4</b>	<b>Installing Jenkins</b>	<b>16</b>
<b>5</b>	<b>Configure Jenkins to publish a container into the registry</b>	<b>16</b>

## 1 Overview

### 1.1 Purpose

This paper documents the Proof of Concept Project: **Enable a Jenkins build server to publish release artifacts as *Docker* images to a *Docker* reegistry. Start service on a new EC2 node by fetching the artifact from the registry.**

## 1.2 Author

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


## 1.3 *Docker* version

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



## 1.4 Deployment workflow

One major issue in Continuous Integration is to insure a once deployed build artifact never changes in future deployment scenarios. *Docker* versions each container, making it immutable after a **docker build**. To start a particular container, being pulled out of a repository, results in starting the same build artifact in all future deployment scenarios. We use the *Jenkins* server to build a docker container triggered by a git push. This container is then pulled and started by the production environment.

### Typical workflow



- code in local environment  
(« dockerized » or not)
- each push to the git repo triggers a hook
- the hook tells a build server to clone the code and run  
« docker build » (using the Dockerfile)
- the containers are tested (nosetests, Jenkins...),  
and if the tests pass, pushed to the registry
- production servers pull the containers and run them
- for network services, load balancers are updated



## 2 *Docker*

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

### 2.1 TODO explain *docker* terminology: Image, Container, Repository, Registry, Index.

### 2.2 Installation

#### 2.2.1 Kernel options

*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 **flora-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/.
```

#### 2.2.2 Installation by hand

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

```
sudo apt-key adv --keyserver keyserver.ubuntu.com \
--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.

### 2.2.3 Installation 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.

### 2.2.4 Installing 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.

### 2.2.5 Configuration

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

## 2.3 Play with *Docker*

We describe some basic *Docker* commands.

### 2.3.1 Check your *Docker* installation.

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

### 2.3.2 Download a pre-built image

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

### 2.3.3 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
```

### 2.3.4 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
```

### 2.3.5 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
```

### 2.3.6 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)"
```

### 2.3.7 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
```

### 2.3.8 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
```

### 2.3.9 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
```

### 2.3.10 Private Repositories

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

Once a repository has your registry's host name as part of the tag, you can push and pull it like any other repository, but it will not be searchable (or indexed at all) in the Central Index, and there will be no user name checking performed. Your registry will function completely independently from the Central Index.



### 2.3.11 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
```

### 2.3.12 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 ownerships might not get preserved.

### 2.3.13 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.

## 2.4 Build your own 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.

### 2.4.1 Download the script

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

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

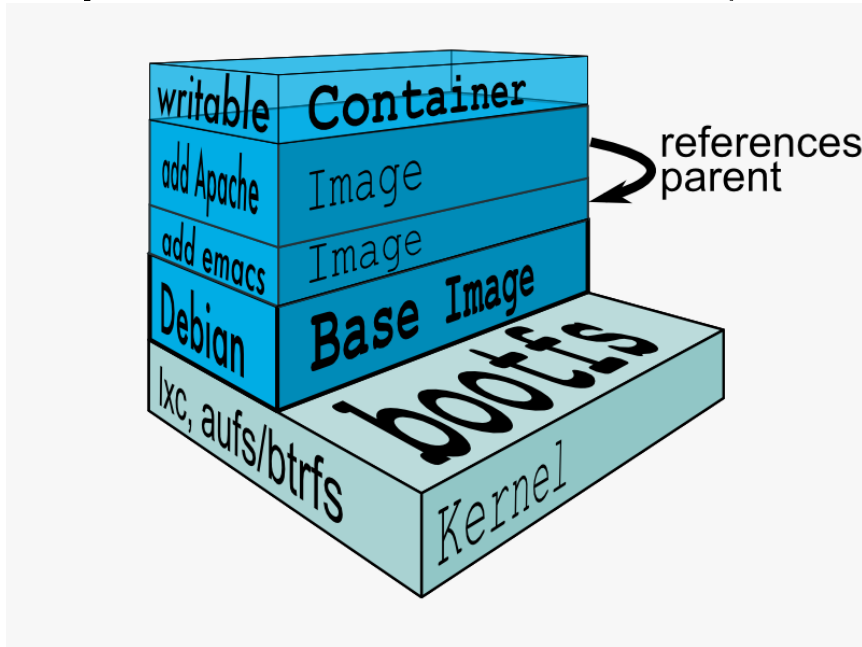
### 2.4.2 Build the base image

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

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

## 2.5 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

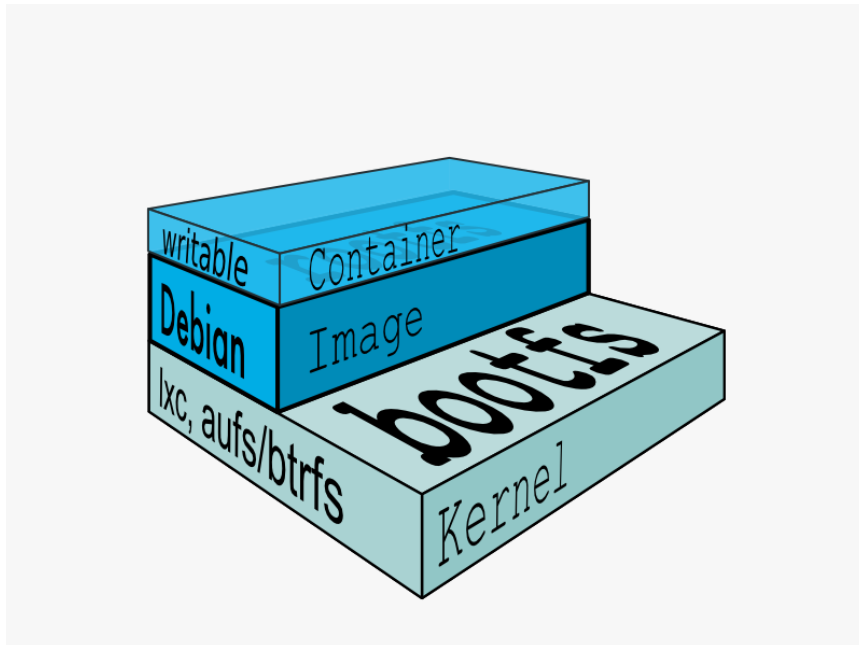


layer.

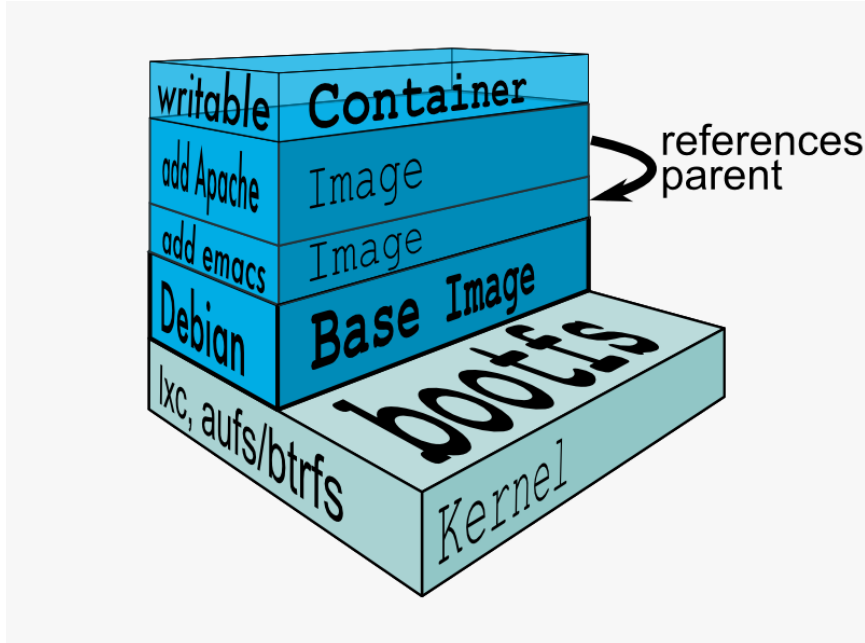
### 2.5.1 Union file system

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**.

### 2.5.2 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.

## 2.6 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`.

### 2.6.1 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*.

### 2.6.2 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`

### 2.6.3 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.

### 2.6.4 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` refers to the user part, the repository part and the tag part of a special image. Note that command `docker images -a` may list the same GUID multiple times as the same image may be tagged differently. Removing an image being tagged multiple times only results in deleting the tag, keeping the other tagged version(s) in the repository.

## 3 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.

### 3.1 Installing the necessary packages and *Java*

We need *jdk* at least version 6, *wget*, *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 may take a minute.

### 3.2 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.

## 3.3 Scala WebApp

### 3.3.1 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
```

### 3.3.2 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*.

### 3.4 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”.

### 3.5 Building a container with the WebApp

#### 3.5.1 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`. Creating directories for each docker file, 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.

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.

### 3.5.2 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
```

## 4 Installing Jenkins

How to install a jenkins server

## 5 Configure Jenkins to publish a container into the registry

Each time the WebApp has changed in git, Jenkins builds a new container, consisting of three parts:

1. Deploying the WebApp-Files into the latest container.
2. Commit the newly build container and tag it properly.
3. Start the newly tagged container.