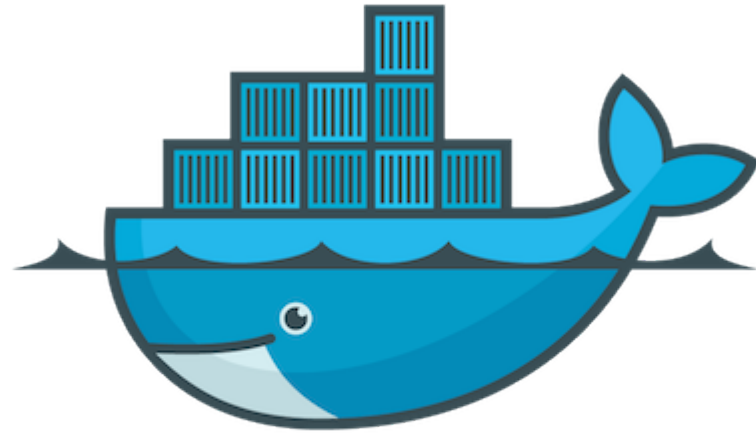




# Docker Fundamentals Exercises

Version: 1.0.1



docker

## Lab 2.1: Hello World

1. Run the `docker run` command.

```
$ sudo docker run busybox echo hello world
```

2. You should see "hello world" returned to your command line.

## Lab 2.2: Pull the ubuntu image

1. Run the `docker pull` command.

```
$ docker pull ubuntu:latest
```

2. Let that run in the background.

## Lab 3.1: Install Docker

1. We've already installed Docker inside our lab environment.
2. Connect to the lab environment.

## Lab 3.2: Stop, start and restart the Docker daemon

We're going to use the `service` command to stop, start and restart the Docker daemon.

1. Stop the Docker daemon.

```
$ service docker stop
```

2. Start the Docker daemon.

```
$ service docker start
```

3. Restart the Docker daemon.

```
$ service docker restart
```

## Lab 3.3: Test the Docker client

1. Use the `docker` client to confirm the Docker daemon is running.

```
$ docker version
```

2. You should see:

```
Client version: 0.11.1
Client API version: 1.11
Go version (client): go1.2.1
Git commit (client): fb99f99
Server version: 0.11.1
Server API version: 1.11
Git commit (server): fb99f99
Go version (server): go1.2.1
Last stable version: 0.11.1
```

## Lab 3.4: See the Docker client help

1. Use the `docker help` command to see what the Docker client can do.

```
$ docker help
```

2. You should see:

```
Usage: docker [OPTIONS] COMMAND [arg...]
-H=[unix:///var/run/docker.sock]: tcp://host:port to bind/connect to or
unix://path/to/socket to use

A self-sufficient runtime for linux containers.

Commands:
  attach  Attach to a running container
  build   Build a container from a Dockerfile
  . . .
```

## Lab 3.5: Don't need no sudo

1. Create the `docker` group.

```
$ sudo groupadd docker
```

2. Add your user to the `docker` group.

```
$ sudo gpasswd -a $USER docker
```

3. Restart the Docker daemon.

```
$ sudo service docker restart
```

4. Log out.

```
$ exit
```

5. Login.

```
$ ssh ec2-user@yourlabhost  
you@ip.com's password:
```

6. Test that Docker works without `sudo`

```
$ docker run ubuntu echo hello world
```



## Lab 4.1: Register an account with Docker Hub

1. Go to [hub.docker.com](https://hub.docker.com) and fill out the form to register an account.
2. Click the activation link in the e-mail you have been sent to confirm.

## Lab 4.2: Logging in to Docker Hub

1. Login to the Docker Hub on the command line now. You'll need the user name and password you created in Lab 1.1.

```
$ docker login
```

2. You should see and type:

```
Username: myDockerHubusername
Password:
Email: my@email.com
Login Succeeded
```

3. Check the contents of the `~/ .dockercfg` file.

```
$ cat ~/.dockercfg
{
  "https://index.docker.io/v1/": {
    "auth": "amFtdHVyMDE6aTliMUw5ckE=",
    "email": "education@docker.com"
  }
}
```

## Lab 4.3: Set up a Github account

If you don't have a Github account, go ahead and set one up now. We will need to use this later for creating Automated Builds, a feature of Docker Hub that allows you to automatically re-build an image when the source code contained within that image changes.



## Lab 5.1: Find an image

1. Run the `docker search` command.

```
$ docker search training
```

2. You should see something like:

NAME	DESCRIPTION	STARS	OFFICIAL
AUTOMATED			
training/jenkins		0	
[OK]			
training/webapp		0	
[OK]			
training/ls		0	
[OK]			
training/namer		0	
[OK]			
training/postgres		0	
[OK]			
training/notes		0	
[OK]			
training/docker-fundamentals-image		0	
[OK]			
training/showoff		0	
. . .			

## Lab 5.2: Retrieve a user image

1. Pull down the `training/docker-fundamentals-image` image using the `docker pull` command.

```
$ docker pull training/docker-fundamentals-image
Pulling repository training/docker-fundamentals-image
8144a5b2bc0c: Pulling dependent layers
511136ea3c5a: Download complete
8abc22fbb042: Download complete
```

## Lab 5.3: Show currently installed images

1. View the currently installed images using the `docker images` command.

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
training/docker-fundamentals-image	latest	8144a5b2bc0c	5 days ago	835 MB
ubuntu	13.10	9f676bd305a4	7 weeks ago	178 MB
ubuntu	saucy	9f676bd305a4	7 weeks ago	178 MB
ubuntu	raring	eb601b8965b8	7 weeks ago	166.5 MB
ubuntu	13.04	eb601b8965b8	7 weeks ago	166.5 MB
ubuntu	12.10	5ac751e8d623	7 weeks ago	161 MB
ubuntu	quantal	5ac751e8d623	7 weeks ago	161 MB
ubuntu	10.04	9cc9ea5ea540	7 weeks ago	180.8 MB
ubuntu	lucid	9cc9ea5ea540	7 weeks ago	180.8 MB
ubuntu	12.04	9cd978db300e	7 weeks ago	204.4 MB
ubuntu	latest	9cd978db300e	7 weeks ago	204.4 MB
ubuntu	precise	9cd978db300e	7 weeks ago	204.4 MB

## Lab 5.4: Retrieve a tagged image

1. Pull down a tagged image using the `docker pull` command.

```
$ docker pull debian:jessie
Pulling repository debian
58394af37342: Download complete
511136ea3c5a: Download complete
8abc22fbb042: Download complete
```

## Lab 6.1: Create a container

1. Sign onto your Amazon EC2 Instance
2. Create a container using the `docker run` command.
3. Execute:

```
$ docker run -t -i ubuntu:14.04 /bin/bash
```

4. Take note of the string of numbers returned after the `root@`. This is the container short ID.



## Lab 6.2: Explore your container

1. Inside the container, print the host system name and kernel version with the `uname` command.

```
root@268e59b5754c:/# uname -rn
268e59b5754c 3.10.40-50.136.amzn1.x86_64
```

2. Now let's exit the container and run that command again.

```
root@268e59b5754c:/# exit
$ uname -rn
ip-172-31-47-238.ec2.internal 3.10.40-50.136.amzn1.x86_64
```

Note that the hostname is different, but the kernel is the same.

## Lab 6.3: What happened to my container?

1. See the status of your container with the `docker ps` command.

```
$ docker ps
```

2. There should be no containers listed. Just the empty titles:

```
CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS  NAMES
```

3. Let's instead try `docker ps` with the `-l` flag.

```
$ docker ps -l
```

4. You will see the information for the last run container, something like:

```
CONTAINER ID  IMAGE          COMMAND          CREATED          STATUS  PORTS  NAMES
a2d4b003d7b6  ubuntu:14.04  /bin/bash       5 minutes ago   Exit 0           sad_pare
```

5. Now run `docker ps` with the `-a` flag.

```
$ docker ps -a
```

6. You should see something like:

```
CONTAINER ID  IMAGE          COMMAND          CREATED          STATUS  PORTS  NAMES
a2d4b003d7b6  ubuntu:14.04  /bin/bash       5 minutes ago   Exit 0           sad_pare
```

## Lab 6.4: What's inside my container?

1. Inspect your container with the `docker inspect` command. We pipe the output of this command into the UNIX `less` command for pagination.

```
$ docker inspect $(docker ps -l -q) | less
```

2. You should see some output much like:

```
[{
  "ID": "<yourContainerID>",
  "Created": "2014-03-15T22:05:42.73203576Z",
  "Path": "/bin/bash",
  "Args": [],
  "Config": {
    "Hostname": "<yourContainerID>",
    "Domainname": "",
    "User": "",
    . . .
  }
}]
```

## Lab 6.5: What's something specific in our container?

Let's inspect a specific aspect of the container.

1. Run `docker inspect` with the `--format` flag and the path to the desired property (as a Go language template) to access a specific property. For instance:

```
$ docker inspect --format='{{ .State.Running }}' <yourContainerID>
```

## Lab 6.6: Starting and attaching to a stopped container

1. Run:

```
$ docker start <yourContainerId>
```

2. This will start the stopped container. In order to attach to it:

```
$ docker attach <yourContainerId>
```

3. You can detach from a container you are attached to using keyboard shortcut.
4. You can also use `docker start -a <yourContainerId>` as a shortcut for "Start, then Attach" to a container.

## Lab 6.7: Stop your container

1. Use the `docker stop` command to stop the container.

```
$ docker stop <yourContainerId>
```

2. Use the `docker rm` command to remove the container.

```
$ docker rm <yourContainerId>
```

## Lab 7.1: Create a new container and make some changes

To start, launch a terminal inside of a `ubuntu` container in interactive mode:

```
$ docker run -it ubuntu bash
root@<yourContainerId>:~/
```

Run the command `apt-get update` to refresh the list of packages available to install, then run the command `apt-get install -y cmatrix` to install the program we are interested in.

```
root@<yourContainerId>:~/ apt-get update && apt-get install -y cmatrix
.... OUTPUT OF APT-GET COMMANDS ....
```

## Lab 7.2: Exit out of the container and inspect the changes

Grab the container's ID and pass it to `docker diff` command as an argument:

```
$ docker ps -lq
<yourContainerId>
$ docker diff <yourContainerId>
C /root
A /root/.bash_history
C /tmp
C /usr
C /usr/bin
A /usr/bin/cmatrix
```



## Lab 7.3: Note that Docker has tracked the changes

Docker keeps track of every file which has changed (C), been added (A), or deleted (D) from the base image used to create the container. For instance, in this container `.bash_history` has been created (since we executed commands at the bash prompt) and many files related to packaging have either been created or modified.

We can formalize these changes into an entirely new image using `docker commit`. The changes will appear as a new layer on top of the existing ones from the base image.

## Lab 7.4: Commit your image

To create an image with a new layer reflecting what appeared in the diff, use `docker commit <containerId>`.

```
$ docker commit <yourContainerId>  
<newImageId>
```

The output of the `docker commit` command will be the ID for your newly created image. You can run:

```
$ docker run -it <newImageId> cmatrix
```

## Lab 7.5: Tagging images

To tag an image, use the `docker tag` command.

```
$ docker tag <newImageId> <dockerhubUsername>/cmatrix
```

Note that `docker commit` also accepts an additional argument if you want to include a tag with your `docker commit` command:

```
$ docker commit <containerId> <tagName>
```

## Lab 7.6: Using image and viewing history

As mentioned, you can use a tagged image like so:

```
$ docker run -it <dockerhubUsername>/cmatrix
```

If you are curious, you can view all of the layers composing an image (and their size, when they were created, etc.) with the `docker history` command:

```
$ docker history ubuntu
IMAGE          CREATED          CREATED BY
c3d5614fecc4   8 days ago      /bin/sh -c #(nop) CMD [/bin/bash]
96e1c132acb3   8 days ago      /bin/sh -c apt-get update && apt-get dist-upg
311ec46308da   8 days ago      /bin/sh -c sed -i 's/^#\s*\s*(deb.*universe\)$/
1b2af7d5307a   8 days ago      /bin/sh -c rm -rf /var/lib/apt/lists/*
c31865d83ea1   8 days ago      /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic
8cbdf71a8e7f   8 days ago      /bin/sh -c #(nop) ADD file:c0f316fa0dcbdd4635
511136ea3c5a   16 months ago
```

## Lab 8.1: Building your own Dockerfile

1. Create a directory to hold our Dockerfile.

```
$ mkdir web_image
```

2. Create a Dockerfile inside this directory.

```
$ cd web_image  
$ touch Dockerfile
```

3. Edit our Dockerfile.

```
$ vim Dockerfile
```

## Lab 8.2: Adding FROM and MAINTAINER instructions

1. Let's start with a FROM instruction.

```
FROM ubuntu:14.04
```

2. Add a MAINTAINER instruction.

```
MAINTAINER Your Name <your@email.com>
```

## Lab 8.3: Installing some packages to support our application

1. Update, install some required packages, and add a simple site to be served, using the `RUN` instruction.

```
RUN apt-get update
RUN apt-get install -y nginx
RUN echo 'Hi, I am in your container' \
    >/usr/share/nginx/html/index.html
```

## Lab 8.4: Install our web application.

1. Set the `CMD` instruction using the array notation. This will be run when the container is started.

```
CMD ["nginx", "-g", "daemon off;"]
```

2. Lastly, use `EXPOSE` to expose port 80.

```
EXPOSE 80
```



## Lab 8.5: Building our web application image.

1. Use `docker build` to build our image.

```
$ docker build -t <dockerhubUsername>/webapp .
```

2. Review the build image with `docker image`.

```
$ docker image <dockerhubUsername>/webapp
REPOSITORY          TAG          IMAGE ID        CREATED         VIRTUAL SIZE
<dockerhubUsername>/webapp latest      e2a9fac29d86   12 seconds ago 246.8 MB
```

## Lab 8.6: Overriding the CMD instruction

1. Specify a command to run instead of the CMD defined when doing `docker run`:

```
docker run -t -t <dockerhubUsername>/web /bin/bash
```

This overrides the default CMD.

## Lab 8.7: Overriding the ENTRYPOINT instruction

1. Specify a command to run instead of the `ENTRYPOINT` defined in the Dockerfile.  
Without `--entrypoint`:

```
$ docker run -t -i training/ls
bin    dev  home  lib64  mnt  proc  run   srv   tmp   var
boot  etc  lib   media  opt  root /sbin  sys   usr
```

With `--entrypoint`:

```
$ docker run --entrypoint /bin/bash -t -i training/ls
root@d902fb7b1fc7:/#
```

## Lab 8.8: Running a container from our image

1. Now let's run a container from our new image using the `docker run` command. The `-P` automatically maps all ports exposed in an image to random ports on the host.

```
$ docker run -d -P <dockerhubUsername>/webapp
```

2. Now checkout our running container using the `docker ps` command. The `-l` flag returns the last container created.

```
$ docker ps -l
```

3. You should see something like:

```
CONTAINER ID  IMAGE                                COMMAND                                CREATED
STATUS        PORTS                                NAMES
25af0d10060f  <dockerhubUsername>/webapp:latest  python app.py 2 minutes ago
Up 2 minutes  0.0.0.0:49162->5000/tcp  tender_newton
```

Make a note of the port number port 5000 is being redirected to.

## Lab 8.9: Using our web application

1. Open a browser and browse to the redirected port on the Docker host.

```
http://<yourHostIP>:redirectedport
```

2. You should see something like.



## Lab 9.1: Push your web image

1. Run the `docker push` command.

```
$ docker push <dockerhubUsername>/web
```

2. Login into your Docker Hub account using your user name, password and email.
3. Confirm your image was successfully pushed.

## Lab 9.2: Review your pushed image

1. Browse to the Docker Hub.

```
https://hub.docker.com/
```

2. Click on the Login link.

```
https://hub.docker.com/account/login/
```

3. Login using your user name and password.

## Lab 9.3: View your image repository

1. Browse to your new image repository, either by clicking on the link or by visiting the following URL:

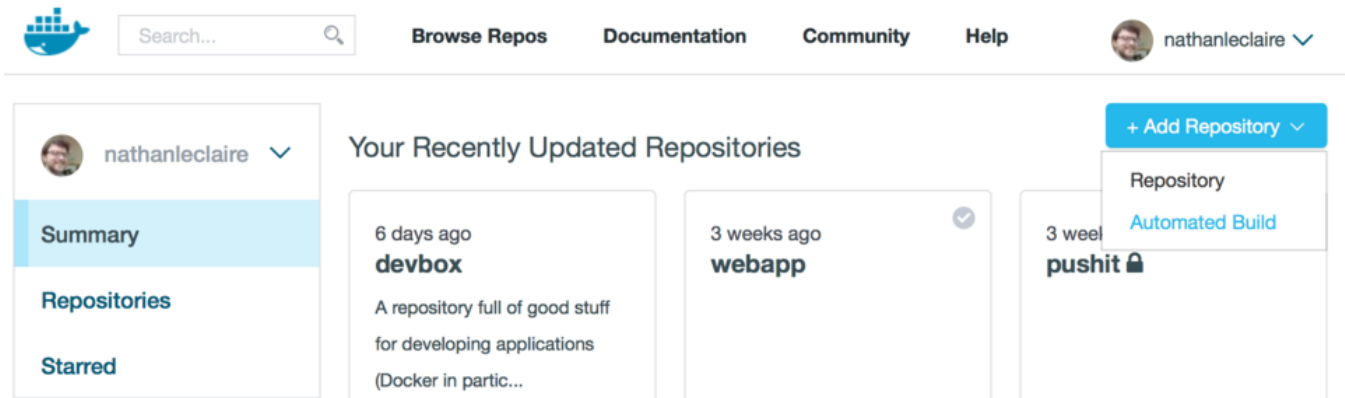
```
https://registry.hub.docker.com/u/<dockerhubUsername>/staticweb/
```

2. Explore the tabs and settings available to you.



## Lab 9.4: Create an automated build

To create an *Automated Build* click on the `Add Repository` button on your main account screen and select `Automated Build`.



# Connecting to your Github account

If this is your first *Automated Build* you will be prompted to connect your GitHub account to the Docker Hub.

Select the source you want to use for your Automated Build



**GitHub**

Select



**Bitbucket**

Select

# Connecting to your Github account

You can then select a specific GitHub repository.

It must contain a `Dockerfile`.

## GitHub: Add Automated Build

For more information on Automated Builds, please read the [Automated Build documentation](#).

Select a Repository to build

 <a href="#">nathanleclaire</a>	
nathanleclaire/twilio_toy_app	Select
nathanleclaire/unix-command-survey	Select
nathanleclaire/wakeup	Select
nathanleclaire/webapp	Select

If you don't have a repository with a `Dockerfile`, you can fork <https://github.com/docker-training/staticweb>, for instance.

# Configuring an automated build

You can then configure the specifics of your *Automated Build* and click the `Create Repository` button.

**README.md**

If you have a README.md file in your repository, we will use that as the repository full description. We will look for the README.md in the same directory where your Dockerfile lives.

Warning: if you change the full description after a build, it will be rewritten the next time the Automated Build, has been built. To make changes, change the README.md in the source repo. For more information please read the [Automated Build documentation](#).

**Namespace (optional) and Repository Name**

nathanleclaire

/

webapp

New unique Repo name; 3 - 30 characters. Only lowercase letters, digits and \_ - . characters are allowed

**Tags**

Type	Name	Dockerfile Location	Docker Tag Name
Branch	master	/	latest

☒ **Public**  
Anyone can pull, and is listed and searchable on the docker index.

☐ **Private**  
Only you can pull, and is not listed on the docker index.

**Active**

☒ When active we will build when new pushes occur

Create Repository

## Lab 10.1: Pull down our application image

1. Pull down our application image.

```
$ docker pull training/namer
```

## Lab 10.2: Download the application's source code

1. Download the application source code.

```
$ git clone https://github.com/docker-training/namer.git
```

2. Review the contents of the `namer` directory.

```
$ cd namer  
$ ls
```

## Lab 10.3: Create a container from our application image.

1. Create a new container (make sure that you are in `namer` directory first).

```
$ docker run -d -v $(pwd):/opt/namer -w /opt/namer \
-p 80:9292 training/namer rackup
```

2. Check the container is running.

```
$ docker ps
```

## Lab 10.4: Update the running application

1. Use your local browser to check the running application.

```
http://<yourHostIP>
```

2. Edit the `namer/company_name_generator.rb` file.

```
$ vim namer/company_name_generator.rb
```

3. Change the `color` CSS property in the `<style>` block from `royalblue` to `red`.

```
<style>
h1, h2 {
  font-family: Georgia, Times New Roman, Times, serif;
  color: red;
  margin: 0;
}
</style>
```

4. Refresh the browser to see the color change.

```
http://<yourHostIP>
```



## Lab 10.5: Stop and remove the container

1. Stop the running container.

```
$ docker stop <yourContainerId>
```

2. Remove the container.

```
$ docker rm <yourContainerId>
```

# Lab 11.1: Create a container

1. Create a new container.

```
$ docker run -d -p 80 training/webapp python -m SimpleHTTPServer 80
```

2. Make a note of the container ID.
3. Check the container is running.

```
$ docker ps
```

## Lab 11.2: Checking the container's port mapping

1. Retrieve the container's port mapping.

```
$ docker port <yourContainerId> 80
```

2. Note also that you can get this information using `docker inspect -f:`

```
$ docker inspect -f "{{ json .NetworkSettings.Ports }}" <yourContainerID>  
{ "5000/tcp":null,"80/tcp":[{"HostIp":"0.0.0.0","HostPort":"49153"}]}
```

3. Make a note of the port number returned.

## Lab 11.3: Browse to the web server

1. Browse to the URL.

```
http://<yourHostIP>:<portNumber>
```

2. You should see a directory listing for your container.

## Lab 11.4: Finding the container's IP address

1. Now retrieve the container's IP address.

```
$ docker inspect --format \
  '{{ .NetworkSettings.IPAddress }}' \
  <yourContainerId>
```

2. Make a note of the IP address returned.
3. Ping the IP address.

```
$ ping <ipAddress>
```

4. You should see a response.

```
64 bytes from <ipAddress>: icmp_req=2 ttl=64 time=0.085 ms
64 bytes from <ipAddress>: icmp_req=2 ttl=64 time=0.085 ms
```

## Lab 12.1: Sharing volumes across containers.

1. Create a container with a volume.

```
$ docker run --name alpha -t -i -v /var/log ubuntu bash
root@<yourContainerID>:/# date >/var/log/now
```

2. Start another container with the same volume. Note the `--volumes-from` flag.

```
$ docker run --volumes-from alpha ubuntu cat /var/log/now
Fri May 30 05:06:27 UTC 2014
```

## Lab 12.2: No-op data containers.

Try out making a data container with a no-op command such as `true`.

```
$ docker run --name wwwdata -v /var/lib/www busybox true
$ docker run --name wwwlogs -v /var/log/www busybox true
```

## Lab 12.3: Sharing a directory between the host and a container

```
$ cd
$ mkdir bindthis
$ ls bindthis
$ docker run -t -i -v $(pwd)/bindthis:/var/www/html/webapp ubuntu bash
root@<yourContainerID>:/# touch /var/www/html/webapp/index.html
root@<yourContainerID>:/# exit
$ ls boundmount_demo
index.html
```



## Lab 12.4: Chaining container volumes together.

1. Create an initial container.

```
$ docker run -t -i -v /var/appvolume --name appdata ubuntu bash
root@<yourContainerID>#
```

2. Create some data in our data volume.

```
root@<yourContainerID># cd /var/appvolume
root@<yourContainerID># echo "Hello" > data
```

3. Exit the container.

```
root@<yourContainerID># exit
```

## Lab 12.5: Chaining container volumes together.

1. Create a new container.

```
$ docker run -t -i --volumes-from appdata --name appserver1 ubuntu bash
root@<yourContainerID>#
```

2. Let's view our data.

```
root@<yourContainerID># cat /var/appvolume/data
Hello
```

3. Let's make a change to our data.

```
root@<yourContainerID># echo "Good bye" \
>> /var/appvolume/data
```

4. Exit the container.

```
root@<yourContainerID># exit
```

## Lab 12.6: Chain containers with data volumes

1. Create a third container.

```
$ docker run -t -i --volumes-from appserver1 \  
  --name appserver2 ubuntu bash  
root@179c063af97a#
```

2. Let's view our data.

```
root@179c063af97a# cat /var/appvolume/data  
Hello  
Good bye
```

3. Exit the container.

```
root@179c063af97a# exit
```

4. Tidy up your containers.

```
$ docker rm appdata appserver1 appserver2
```

## Lab 12.7: Sharing a single file between the host and a container

```
$ echo 4815162342 > /tmp/numbers
$ docker run -t -i -v /tmp/numbers:/numbers ubuntu bash
root@274514a6e2eb:/# cat /numbers
4815162342
```

## Lab 12.8: Sharing a socket and docker binary with a container

```
$ docker run -t -i -v /var/run/docker.sock:/var/run/docker.sock -v $(which
docker):/docker ubuntu ./docker
Usage: docker [OPTIONS] COMMAND [arg...]

-H=[unix:///var/run/docker.sock]: tcp://host:port to bind/connect to or unix://path/to/
socket to use

A self-sufficient runtime for linux containers.

....
```

Be careful: if a container can access `/var/run/docker.sock`, it will be able to do *anything it wants* on the host!

## Lab 13.1: Pull the `redis` image

```
$ docker pull redis:latest
```

```
$ docker images redis
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
redis	latest	b73cdc045d3c	2 weeks ago	98.42 MB

## Lab 13.2: Start a Redis container

```
$ docker run -d --name mycache redis
<yourContainerId>
$ docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	
STATUS	PORTS	NAMES		
<yourContainerId>	redis:latest	redis-server	5 seconds ago	Up 4
seconds	6379/tcp	mycache		

## Lab 13.3: Pull Rails application image

```
$ docker pull nathanleclaire/redisonrails
```

When it's done downloading, review it.

```
$ docker images nathanleclaire/redisonrails
```



## Lab 13.4: Try starting Rails container without links

Try interacting with `$redis` in the rails container without links. It will fail.

```
$ docker run -t -i nathanleclaire/redisonrails rails console
Loading development environment (Rails 4.0.2)
irb(main):001:0> $redis
=> #<Redis client v3.1.0 for redis://redis:6379/0>
irb(main):002:0> $redis.set('foo', 'bar')
SocketError: getaddrinfo: Name or service not known
    from /usr/local/lib/ruby/gems/2.1.0/gems/redis-3.1.0/lib/redis/connection/
ruby.rb:152:in `getaddrinfo'
    from /usr/local/lib/ruby/gems/2.1.0/gems/redis-3.1.0/lib/redis/connection/
ruby.rb:152:in `connect'
    from /usr/local/lib/ruby/gems/2.1.0/gems/redis-3.1.0/lib/redis/connection/
ruby.rb:211:in `connect'
    from /usr/local/lib/ruby/gems/2.1.0/gems/redis-3.1.0/lib/redis/client.rb:304:in
`establish_connection'
    ....
```

## Lab 13.5: Start Rails container with links

Try interacting with `$redis` *with* links. It will work.

```
$ docker run -i -t --link mycache:redis \
  nathanleclaire/redisonrails rails console
Loading development environment (Rails 4.0.2)
irb(main):001:0> $redis
=> #<Redis client v3.1.0 for redis://redis:6379/0>
irb(main):002:0> $redis.set('a', 'b')
=> "OK"
irb(main):003:0> $redis.get('a')
=> "b"
irb(main):004:0> $redis.set('someHash', {:foo => 'bar', :spam => 'eggs'})
=> "OK"
irb(main):005:0> $redis.get('someHash')
=> "{:foo=>\"bar\", :spam=>\"eggs\"}"
irb(main):006:0> $redis.set('users', ['Aaron', 'Jerome', 'Nathan'])
=> "OK"
irb(main):007:0> $redis.get('users')
=> "[\"Aaron\", \"Jerome\", \"Nathan\"]"
irb(main):008:0> exit
```

## Lab 13.6: See links environment variables

```
$ docker run --link mycache:redis nathanleclaire/redisonrails env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=0738e57b771e
REDIS_PORT=tcp://172.17.0.120:6379
REDIS_PORT_6379_TCP=tcp://172.17.0.120:6379
REDIS_PORT_6379_TCP_ADDR=172.17.0.120
REDIS_PORT_6379_TCP_PORT=6379
REDIS_PORT_6379_TCP_PROTO=tcp
REDIS_NAME=/dreamy_wilson/redis
REDIS_ENV_REDIS_VERSION=2.8.13
REDIS_ENV_REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-2.8.13.tar.gz
REDIS_ENV_REDIS_DOWNLOAD_SHA1=a72925a35849eb2d38a1ea076a3db82072d4ee43
HOME=/
RUBY_MAJOR=2.1
RUBY_VERSION=2.1.2
```

## Lab 13.7: See links DNS entry work

```
$ docker run -it --link mycache:redis nathanleclaire/redisonrails ping redis
PING redis (172.17.0.29): 56 data bytes
64 bytes from 172.17.0.29: icmp_seq=0 ttl=64 time=0.164 ms
64 bytes from 172.17.0.29: icmp_seq=1 ttl=64 time=0.122 ms
64 bytes from 172.17.0.29: icmp_seq=2 ttl=64 time=0.086 ms
^C--- redis ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.086/0.124/0.164/0.032 ms
```

## Lab 13.8: Start our linked Rails application

```
$ docker run -d -p 80:3000 --link mycache:redis nathanleclaire/redisonrails
```

To verify that it is running:

```
$ docker ps -l
```

## Lab 13.9: Visit our linked Rails application

Finally, let's browse to our application and confirm it's working.

```
http://<yourHostIP>
```



**This is an app connected to Redis.**

It has been viewed 40 times.

# Lab 14.1: Installing fig

To install `fig`:

```
curl -L https://github.com/docker/fig/releases/download/0.5.2/linux \
> /usr/local/bin/fig
chmod +x /usr/local/bin/fig
```

You can also use `pip` if you prefer:

```
sudo pip install -U fig
```

## Lab 14.2: Clone repo

Clone the source code for the app we will be working on.

```
cd  
git clone https://github.com/docker-training/simplefig  
cd simplefig
```



## Lab 14.3: Dockerfile

Create this Dockerfile:

```
FROM python:2.7
ADD requirements.txt /code/requirements.txt
WORKDIR /code
RUN pip install -r requirements.txt
ADD . /code
```

## Lab 14.4: fig.yml

Now create a `fig.yml` to store the runtime properties of the app.

```
web:
  build: .
  command: python app.py
  ports:
    - "5000:5000"
  volumes:
    - ./code
  links:
    - redis
redis:
  image: orchardup/redis
```

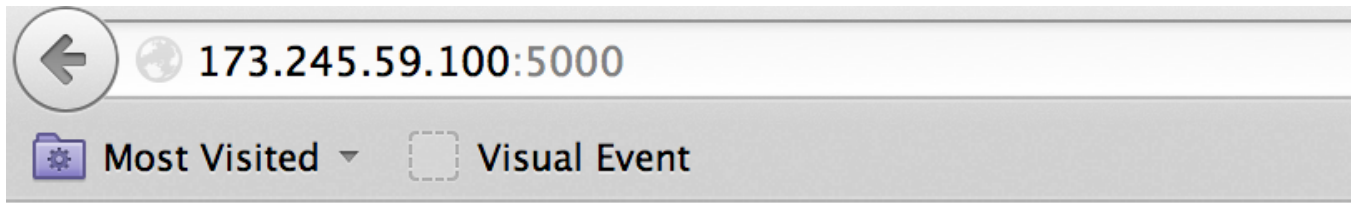
## Lab 14.5: Fig up

Run `fig up` in the directory, and watch fig build and run your app with the correct parameters, including linking the relevant containers together.

```
fig up
```

## Lab 14.6: Verify app is running

Verify that the app is running at `http://<yourHostIP>:5000`.



Hello Docker Training! I have been seen 8 times

## Lab 14.7: Additional figlets

`fig` introduces a unit of abstraction called a "service" (mostly, a container that interacts with other containers in some way and has specific runtime properties).

To rebuild all the services in your `fig.yml`:

```
fig build
```

To run `fig up` in the background instead of the foreground:

```
fig up -d
```

To see currently running services:

```
fig ps
```

To remove the existing services:

```
fig rm
```

## Lab 16.1: Create a GitHub user

1. You'll need a GitHub user to complete this section.
2. If you already have a GitHub user please sign into GitHub.

```
https://github.com/login
```

3. If you don't have a GitHub account please sign up for a free account here:

```
https://github.com/join
```

## Lab 16.2: Link Docker Hub to GitHub

1. Sign into the Docker Hub.

```
https://index.docker.io/
```

2. Link your Docker Hub account to GitHub.

```
https://index.docker.io/
```

## Lab 16.3: Fork our training/webapp repository

1. Browse to our training/webapp repository.

```
https://github.com/docker-training/webapp
```

2. Click the fork button.



## Lab 16.4: Add an Automated Build

1. Go to your Docker Hub profile page and click `Add automated (source) build`
2. Click on `GitHub` for the service you want to use.
3. We then select the `webapp` repository from your GitHub Account.

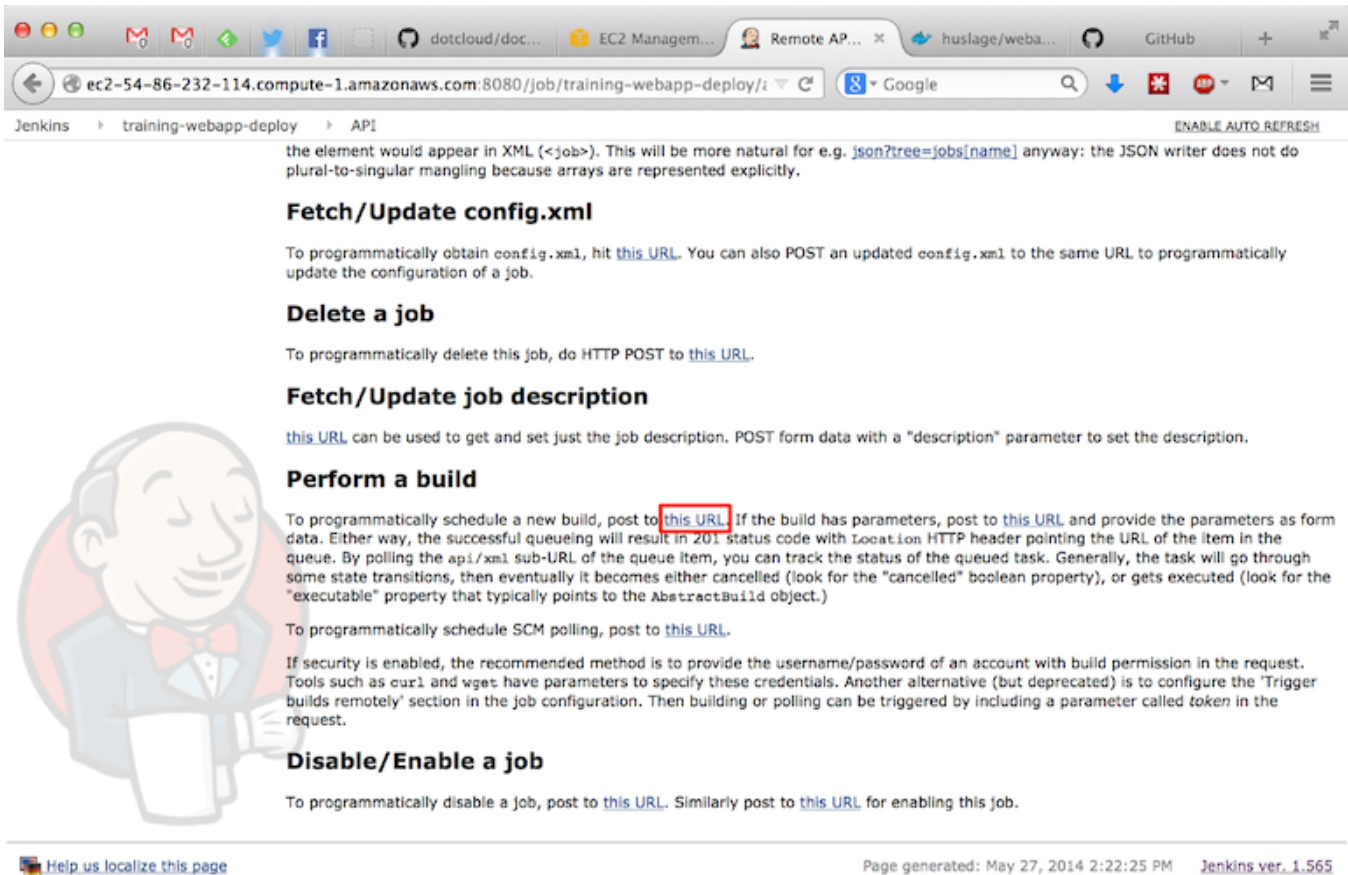
*Make sure to un-check the `When active we will build when new pushes occur` box.*

## Lab 16.5: Start Jenkins

1. Start a Jenkins instance in our container.  
**@@@** Sh \$ docker run -d --name=jenkins -p 8080:8080 \ -v /var/run/docker.sock:/var/run/docker.sock \ -e DOCKERHUBID=*hub id*> \ -e DOCKERHUBEMAIL=*hub email*> \ -e GITHUB\_ID=*id*> \ nathanleclaire/jenkins

# Lab 16.6: Find the Webhook

1. From the Jenkins Dashboard click the `training-webapp-deploy` job. In the bottom-right of the page you will see a link for the REST API. Click that link.
2. Scroll down a bit to the `Perform a build` section.
3. **Copy** the link where it says `Post to this URL`.



the element would appear in XML (<job>). This will be more natural for e.g. `json?tree=jobs[name]` anyway: the JSON writer does not do plural-to-singular mangling because arrays are represented explicitly.

### Fetch/Update config.xml

To programmatically obtain `config.xml`, hit [this URL](#). You can also POST an updated `config.xml` to the same URL to programmatically update the configuration of a job.

### Delete a job

To programmatically delete this job, do HTTP POST to [this URL](#).

### Fetch/Update job description

[this URL](#) can be used to get and set just the job description. POST form data with a "description" parameter to set the description.

### Perform a build

To programmatically schedule a new build, post to [this URL](#). If the build has parameters, post to [this URL](#) and provide the parameters as form data. Either way, the successful queuing will result in 201 status code with `Location` HTTP header pointing the URL of the item in the queue. By polling the `api/xml` sub-URL of the queue item, you can track the status of the queued task. Generally, the task will go through some state transitions, then eventually it becomes either cancelled (look for the "cancelled" boolean property), or gets executed (look for the "executable" property that typically points to the `AbstractBuild` object.)

To programmatically schedule SCM polling, post to [this URL](#).

If security is enabled, the recommended method is to provide the username/password of an account with build permission in the request. Tools such as `curl` and `wget` have parameters to specify these credentials. Another alternative (but deprecated) is to configure the 'Trigger builds remotely' section in the job configuration. Then building or polling can be triggered by including a parameter called `token` in the request.

### Disable/Enable a job

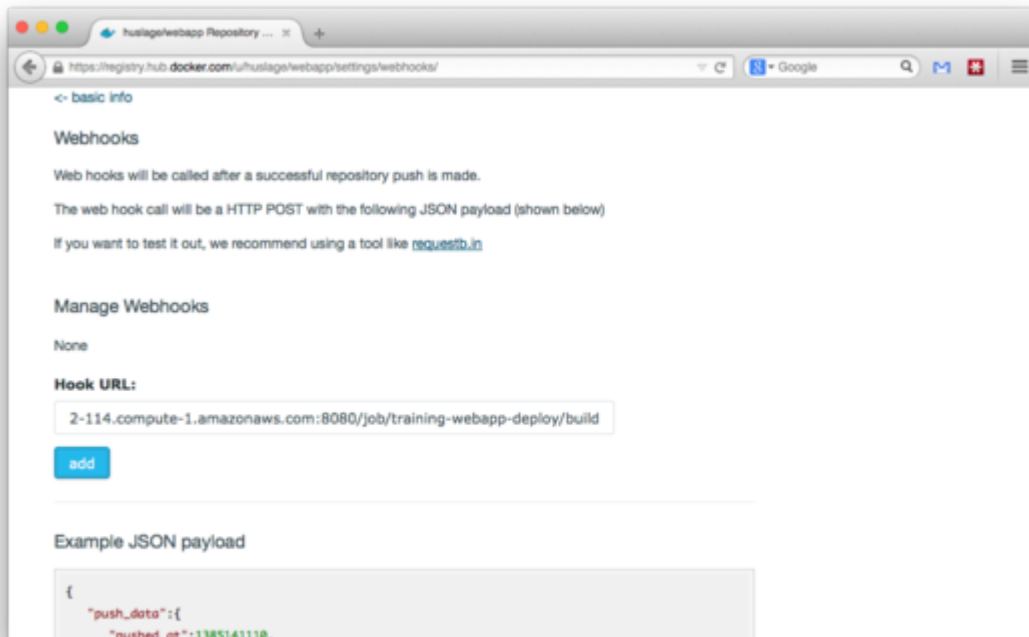
To programmatically disable a job, post to [this URL](#). Similarly post to [this URL](#) for enabling this job.

[Help us localize this page](#)

Page generated: May 27, 2014 2:22:25 PM [Jenkins ver. 1.565](#)

## Lab 16.7: Configure the Webhook

1. Go back to your `webapp` Docker Hub repository Settings
2. Click `Webhooks` on the left sidebar.
3. **Paste** the URL you just copied into the `Hook URL` box.



## Lab 16.8: Test the job

1. Go to the Jenkins Dashboard. Click the Play icon at the right side of the `training-webapp-build` job.
2. You can watch the progress in the left sidebar as jobs progress.
3. You can also view progress by clicking on the job names and then the job numbers. Click around and have fun.

# Lab 18.1: Set up a CA

1. Initialize the CA in an empty directory.

```
$ mkdir docker-ca
$ chmod 0700 docker-ca
$ cd docker-ca
$ echo 01 > ca.srl
$ openssl genrsa -des3 -out ca-key.pem 2048
$ openssl req -new -x509 -days 365 -key ca-key.pem -out ca.pem
```

## Lab 18.2: Generate Server Key and Sign it

### 1. Generate the Server Key

```
$ openssl genrsa -des3 -out server-key.pem 2048
$ openssl req -subj '/CN=**<Your Hostname Here>**' -new -key server-key.pem
-out server.csr
$ openssl rsa -in server-key.pem -out server-key.pem
```

### 2. Sign the key with our CA:

```
$ openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.pem \
-out server-cert.pem
```

## Lab 18.3: Generate Client Key and Sign it

### 1. Generate the Client Key

```
$ openssl genrsa -des3 -out client-key.pem 2048
$ openssl req -subj '/CN=client' -new -key client-key.pem -out client.csr
$ openssl rsa -in client-key.pem -out client-key.pem
```

### 2. Create an extensions config file

```
$ echo extendedKeyUsage = clientAuth > extfile.cnf
```

### 3. Sign the key

```
$ openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey ca-key.pem \
-out client-cert.pem -extfile extfile.cnf
```



## Lab 18.4: Change Docker startup options

1. Copy the keys from the `docker-ca` directory to `/etc/docker`

```
$ sudo mkdir /etc/docker
$ sudo chown docker:docker /etc/docker
$ sudo chmod 0700 /etc/docker
$ sudo cp ~/docker-ca/{ca,server-key,server-cert}.pem /etc/docker
```

2. Edit the file `/etc/default/docker`

```
$ sudo nano /etc/default/docker
```

3. Change the line:

```
DOCKER_OPTS="-H tcp://127.0.0.1:4243 -H unix:///var/run/docker.sock"
```

to

```
DOCKER_OPTS="-H tcp://127.0.0.1:2376 --tlsverify --tlscacert=/etc/docker/
ca.pem --tlscert=/etc/docker/server-cert.pem --tlskey=/etc/docker/
server-key.pem"
```

and save the file.

1. Restart docker

```
$ sudo service docker restart
```

## Lab 18.5: Set up the client and test.

1. Copy the keys into the `~/ .docker` directory

```
$ mkdir ~/.docker  
$ cp ~/docker-ca/ca.pem ~/.docker  
$ cp ~/docker-ca/client-key.pem ~/.docker/key.pem  
$ cp ~/docker-ca/client-cert.pem ~/.docker/cert.pem
```

2. Test the connection

```
$ docker --tlsverify ps
```

All done!

## Lab 19.1: Test the `info` Docker Remote API Endpoint

1. Go to the command line.
2. Use the `curl` command to test our connection to the local Docker daemon.

```
$ curl --silent -X GET http://localhost:2375/info \
  | python -mjson.tool
```

3. You should see output like:

```
{
  "Containers": 68,
  "Debug": 0,
  "Driver": "aufs",
  "DriverStatus": [
    [
      "Root Dir",
      "/var/lib/docker/aufs"
    ],
    [
      "Dirs",
      "711"
    ]
  ],
  . . .
}
```

## Lab 19.2: Creating a new container via the API

1. Now let's see how to create a container with the API.

```
$ curl -X POST -H 'Content-Type: application/json' \
  http://localhost:2375/containers/create \
  -d '{
    "Cmd":["echo", "hello world"],
    "Image":"busybox"
  }'
```

2. You should see a container ID returned.

```
{"Id":"<yourContainerID>","Warnings":null}
```

## Lab 19.3: Starting our new container

1. Start the container that we just created.

```
$ curl -X POST -H 'Content-Type: application/json' \  
  http://localhost:2375/containers/<yourContainerID>/start \  
  -d '{}'
```

No output will be shown, unless an error happens.

## Lab 19.4: Inspecting our launched container

1. Now inspect our freshly launched container.

```
$ curl --silent \  
http://localhost:2375/containers/<yourContainerID>/json
```

2. You should see similar output to the `docker inspect` command.

```
{  
  "Args": [  
    "hello world"  
  ],  
  "Config": {  
    "AttachStderr": false,  
    "AttachStdin": false,  
    "AttachStdout": false,  
    "Cmd": [  
      "echo",  
      "hello world"  
    ],  
    . . .  
  }  
}
```

## Lab 19.5: Waiting for the container to exit, and check its exit status

1. Call the `wait` endpoint.

```
$ curl --silent -X POST \  
http://localhost:2375/containers/<yourContainerID>/wait
```

2. Check the result.

```
{"StatusCode":0}
```

## Lab 19.6: Viewing container output

1. Call the `logs` endpoint, asking for the `stdout` stream.

```
$ curl --silent \  
http://localhost:2375/containers/<yourContainerID>/logs?stdout=1
```

2. You should see the result of the `echo` command.

```
hello world
```



## Lab 19.7: Working with images

1. Now lets see all the images available on the Docker host.

```
$ curl -X GET http://localhost:2375/images/json?all=0
```

2. You should see a hash of all images on the Docker host.

```
[
  {
    "Created": 1396291095,
    "Id":
    "cccdc2d2ec497e814793e8bd952ae76d5d552c8bb7ed927db54aa65579508ffd",
    "ParentId":
    "9cd978db300e27386baa9dd791bf6dc818f13e52235b26e95703361ec3c94dc6",
    "RepoTags": [
      "training/datavol:latest"
    ],
    "Size": 0,
    "VirtualSize": 204371253
  },
]
```

## Lab 19.8: Searching the Docker Hub for an image

1. Now initiate a search for a specific image.

```
$ curl -X GET http://localhost:2375/images/search?term=training
```

2. You should see a hash of images.

```
[
  {
    "description": "",
    "is_official": false,
    "is_trusted": true,
    "name": "training/namer",
    "star_count": 0
  },
  . . .
]
```

## Lab 19.9: Creating an image

1. Now download one of these images.

```
$ curl -i -v -X POST \  
http://localhost:2375/images/create?fromImage=training/namer
```

2. You should see a status indicating it is retrieving the image.

```
{"status":"Pulling repository training/namer"}
```