Docker is a great tool for deploying your servers. Docker even has a public registry called Docker Hub to store Docker images. While Docker lets you upload your Docker creations to their Docker Hub for free, anything you upload is also public. This might not be the best option for your project.

This guide will show you how to set up and secure your own private Docker registry. By the end of this tutorial you will be able to push a custom Docker image to your private registry and pull the image securely from a different host.

This tutorial doesn't cover containerizing your own application but only how to create the registry where you can store your deployments. If you want to learn how to get started with Docker itself (as opposed to the registry), you may want to read the How To Install and Use Docker: Getting Started tutorial.

This tutorial has been tested with both the registry server and registry client running **Ubuntu 14.04**, but it may work with other Debian-based distributions. It also covers version 2.0 of the Docker Registry.

Docker Concepts

If you haven't used Docker before then it's worth taking a minute to go through a few of Docker's key concepts. If you're already using Docker and just want to know how to get started running your own registry, then please skip ahead to the next section.

For a refresher on how to use Docker, take a look at the excellent Docker Cheat Sheet.

Docker at its core is a way to separate an application and the dependencies needed to run it from the operating system itself. To make this possible Docker uses *containers* and *images*. A Docker image is basically a template for a filesystem. When you run a Docker image, an instance of this filesystem is made live and runs on your system inside a Docker container. By default this container can't touch the original image itself or the filesystem of the host where Docker is running. It's a self-contained environment.

Whatever changes you make in the container are preserved in that container itself and don't affect the original image. If you decide you want to keep those changes, then you can "commit" a container to a Docker image (via the `docker commit` command). This means you can then spawn new containers that start with the contents of your old container, without affecting the original container (or image). If you're familiar with `git`, then the workflow should seem quite similar: you can create new branches (images in Docker parlance) from any container. Running an image is a bit like doing a `git checkout`.

To continue the analogy, running a private Docker registry is like running a private Git repository for your Docker images.

Prerequisites

To complete this tutorial, you will need the following:

- 2 Ubuntu 14.04 Droplets: one for the private Docker registry and one for the Docker client
- A non-root user with sudo privileges on each Droplet (Initial Server Setup with Ubuntu 14.04 explains how to set this up.)
- Docker and Docker Compose installed with the instructions from How To Install and Use Docker Compose on Ubuntu 14.04
- A domain name that resolves to the Droplet for the private Docker registry

Step 1 — Installing Package for Added Security

To set up security for the Docker Registry it's best to use Docker Compose. This way we can easily run the Docker Registry in one container and let Nginx handle security and communication with the outside world in another. You should already have it installed from the Prerequisites section.

Since we'll be using Nginx to handle our security, we'll also need a place to store the list of username and password combinations that we want to access our registry. We'll install the `apache2-utils` package which contains the `htpasswd` utility that can easily generate password hashes Nginx can understand:

```
sudo apt-get -y install apache2-utils
```

Step 2 — Installing and Configuring the Docker Registry

The Docker command line tool works great for starting and managing a Docker container or two, but most apps running inside Docker containers don't exist in isolation. To fully deploy most apps you need a few components running in parallel. For example, most web applications are made up of a web server that serves up the app's code, an interpreted scripting language such as PHP or Ruby (with Rails), and a database server like MySQL.

Docker Compose allows you to write one `.yml` configuration file for the configuration for each container as well as information about how the containers communicate with each other. You then use the `docker-compose` command line tool to issue commands to all the components that make up an application.

Since the Docker registry itself is an application with multiple components, we'll use Docker Compose to manage our configuration.

To start a basic registry the only configuration needed is to define the location where your registry will be storing its data. Let's set up a basic Docker Compose YAML file to bring up a basic instance of the registry.

First create a folder where our files for this tutorial will live and some of the subfolders we'll need:

```
mkdir ~/docker-registry && cd $_
mkdir data
```

Using your favorite text editor, create a `docker-compose.yml` file:

```
nano docker-compose.yml
```

Add the following contents to the file:

docker-compose.yml

```
registry:
  image: registry:2
  ports:
    - 127.0.0.1:5000:5000
  environment:
    REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
  volumes:
    - ./data:/data
```

The interesting bit here is at the end. The `environment` section sets an environment variable in the Docker registry container with the path `/data`. The Docker registry app knows to check this environment variable when it starts up and to start saving its data to the `/data` folder as a result.

Only in this case, the `volumes: - ./data:/data` bit is telling Docker that the `/data` directory in that container should actually map out to `/data` on our host machine. So the end result is that the Docker registry's data all gets stored in `~/docker-registry/data` on our local machine.

Let's go ahead and start it up to make sure everything is in order:

```
cd ~/docker-registry
docker-compose up
```

You'll see a bunch of download bars come move across your screen (this is Docker downloading the actual Docker registry image from Docker's own Docker Registry). If everything went well in a minute or two, you should see output that looks like this (versions might vary):

```
Output of docker-compose up
registry_1 | time="2015-10-18T23:45:58Z" level=warning msg="No HTTP secret provided - generated random secret. This may cause problems with uploads if multiple registries are behind a load-balancer. To provide a shared secret, fill in http.secret in the configuration file or set the REGISTRY_HTTP_SECRET environment variable."
```

```
instance.id=44c828de-c27a-401e-bb2e-38b17e6a4b7b version=v2.1.1
registry_1 | time="2015-10-18T23:45:58Z" level=info msg="redis not configured"
instance.id=44c828de-c27a-401e-bb2e-38b17e6a4b7b version=v2.1.1
registry_1 | time="2015-10-18T23:45:58Z" level=info msg="using inmemory blob descriptor
cache" instance.id=44c828de-c27a-401e-bb2e-38b17e6a4b7b version=v2.1.1
registry_1 | time="2015-10-18T23:45:58Z" level=info msg="listening on [::]:5000"
instance.id=44c828de-c27a-401e-bb2e-38b17e6a4b7b version=v2.1.1
registry_1 | time="2015-10-18T23:45:58Z" level=info msg="Starting upload purge in 1m0s"
instance.id=44c828de-c27a-401e-bb2e-38b17e6a4b7b version=v2.1.1
```

Don't worry about the `No HTTP secret provided` message. It's normal.

Great! At this point you've already got a full Docker registry up and running and listening on port 5000 (this was set by the `ports:` bit in the `docker-compose.yml` file). At this point the registry isn't that useful yet — it won't start unless you bring up the registry manually. Also, Docker registry doesn't come with any built-in authentication mechanism, so it's insecure and completely open to the public right now.

Docker Compose will by default stay waiting for your input forever, so go ahead and hit `CTRL-C` to shut down your Docker registry container.

Step 3 — Setting Up an Nginx Container

Let's get to work on fixing these security issues. The first step is to set up a copy of Nginx inside another Docker container and link it up to our Docker registry container.

Let's start by creating a directory to store our Nginx configuration:

```
mkdir ~/docker-registry/nginx
```

Now, re-open your `docker-compose.yml` file in the `~/docker-registry` directory:

```
nano docker-compose.yml
```

Paste the following into the top of the file:

docker-compose.yml

```
nginx:
  image: "nginx:1.9"
  ports:
    - 5043:443
  links:
    - registry:registry
  volumes:
```

```
    - ./nginx/:/etc/nginx/conf.d:ro
```

This will create a new Docker container based on the official Nginx image. The interesting bit here is the `links` section. It automagically set up a "link" from one Docker container to the another. When the Nginx container starts up, it will be able to reach the `registry` container at the hostname `registry` no matter what the actual IP address the `registry` container ends up having. (Behind the scenes Docker is actually inserting an entry into the `/etc/hosts` file in the `nginx` container to tell it the IP of the `registry` container).

The `volumes:` section is similar to what we did for the `registry` container. In this case it gives us a way to store the config files we'll use for Nginx on our host machine instead of inside the Docker container. The `:ro` at the end just tells Docker that the Nginx container should only have read-only access to the host filesystem.

Your full `docker-compose.yml` file should now look like this:

docker-compose.yml

```
nginx:
  image: "nginx:1.9"
  ports:
    - 5043:443
  links:
    - registry:registry
  volumes:
    - ./nginx/:/etc/nginx/conf.d
registry:
  image: registry:2
  ports:
    - 127.0.0.1:5000:5000
  environment:
    REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
  volumes:
    - ./data:/data
```

Running `docker-compose up` will now start two containers at the same time: one for the Docker registry and one for Nginx.

We need to configure Nginx before this will work though, so let's create a new Nginx configuration file.

Create a `registry.conf` file:

```
nano ~/docker-registry/nginx/registry.conf
```

Copy the following into the file:

~/docker-registry/nginx/registry.conf

```
upstream docker-registry {
  server registry:5000;
}

server {
  listen 443;
  server_name myregistrydomain.com;

  # SSL
  # ssl on;
  # ssl_certificate /etc/nginx/conf.d/domain.crt;
  # ssl_certificate_key /etc/nginx/conf.d/domain.key;

  # disable any limits to avoid HTTP 413 for large image uploads
  client_max_body_size 0;

          #     required     to     avoid     HTTP     411:     see     Issue     #1486
(https://github.com/docker/docker/issues/1486)
  chunked_transfer_encoding on;

  location /v2/ {
    # Do not allow connections from docker 1.5 and earlier
     # docker pre-1.6.0 did not properly set the user agent on ping, catch "Go *" user
agents
    if ($http_user_agent ~ "^(docker\/1\.(3|4|5(?!\.[0-9]-dev))|Go ).*$" ) {
      return 404;
    }

    # To add basic authentication to v2 use auth_basic setting plus add_header
    # auth_basic "registry.localhost";
    # auth_basic_user_file /etc/nginx/conf.d/registry.password;
    # add_header 'Docker-Distribution-Api-Version' 'registry/2.0' always;

    proxy_pass                          http://docker-registry;
    proxy_set_header  Host              $http_host;   # required for docker client's sake
    proxy_set_header  X-Real-IP         $remote_addr; # pass on real client's IP
    proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Proto $scheme;
    proxy_read_timeout                  900;
  }
}
```

Save and exit the file.

Now you can install Nginx and start up the two Docker containers all with one command:

```
docker-compose up
```

Nginx doesn't print any output on startup, but if all went well you're now running a copy of Nginx that is set up to proxy to your `registry` container. To test it, let's use `curl` to make an HTTP request to our Docker registry directly, and then make another request to our Nginx port. If everything is set up correctly the output will be the same in both cases (as of this writing Docker returns an empty json object "`{}`") since Nginx will proxy the request through to the Docker registry.

First, make an HTTP request directly to the Docker registry:

```
curl http://localhost:5000/v2/
```

As of this writing Docker returns an empty json object, so you should see:

```
Output
{}
```

Now send an HTTP request to the Nginx port:

```
curl http://localhost:5043/v2/
```

You should see the same output:

```
Output
{}
```

If things are working correctly you'll see some output in your `docker-compose` terminal that looks like the below as well:

```
Output of docker-compose
        registry_1  |  time="2015-08-11T10:24:53.746529894Z"  level=debug  msg="authorizing
request"             environment=development             http.request.host="localhost:5043"
http.request.id=55c3e2a6-4f34-4b0b-bc57-11c814b4f4d3                http.request.method=GET
http.request.remoteaddr=172.17.42.1                                 http.request.uri="/v2/"
http.request.useragent="curl/7.35.0"       instance.id=55634dfc-c9e0-4ec9-9872-6f4930c17759
service=registry version=v2.0.1
    registry_1 | time="2015-08-11T10:24:53.747650205Z" level=info msg="response completed"
environment=development  http.request.host="localhost:5043"  http.request.id=55c3e2a6-4f34-
4b0b-bc57-11c814b4f4d3     http.request.method=GET     http.request.remoteaddr=172.17.42.1
http.request.uri="/v2/"                                  http.request.useragent="curl/7.35.0"
http.response.contenttype="application/json;                                  charset=utf-8"
```

```
http.response.duration=8.143193ms      http.response.status=200      http.response.written=2
instance.id=55634dfc-c9e0-4ec9-9872-6f4930c17759 service=registry version=v2.0.1
    registry_1 | 172.17.0.21 - - [11/Aug/2015:10:24:53 +0000] "GET /v2/ HTTP/1.0" 200 2 ""
"curl/7.35.0"
    nginx_1    | 172.17.42.1 - - [11/Aug/2015:10:24:53 +0000] "GET /v2/ HTTP/1.1" 200 2 "-"
"curl/7.35.0" "-"
```

If you see lines with the `registry_` prefix (the number after the _ may be different on your machine) then all is well, and Nginx has successfully proxied our HTTP request to the Docker registry.

Go ahead and hit `CTRL-C` again in your `docker-compose` terminal to shut down your Docker containers.

Step 4 — Setting Up Authentication

Now that Nginx is proxying requests properly let's set it up with HTTP authentication so that we can control who has access to our Docker registry. To do that we'll create an authentication file in Apache format (Nginx can read it too) via the `htpasswd` utility we installed earlier and add users to it.

Create the first user as follows, replacing USERNAME with the username you want to use:

```
cd ~/docker-registry/nginx
htpasswd -c registry.password USERNAME
```

Create a new password for this user when prompted.

If you want to add more users in the future, just re-run the above command without the `-c` option (the `c` is for create):

```
htpasswd registry.password USERNAME
```

At this point we have a `registry.password` file with our users set up and a Docker registry available. You can take a peek at the file at any point if you want to view your users (and remove users if you want to revoke access).

Next, we need to tell Nginx to use that authentication file.

Open up `~/docker-registry/nginx/registry.conf` in your favorite text editor:

```
nano ~/docker-registry/nginx/registry.conf
```

Scroll to the middle of the file where you'll see some lines that look like this:

~/docker-registry/nginx/registry.conf

```
# To add basic authentication to v2 use auth_basic setting plus add_header
```

```
# auth_basic "registry.localhost";
# auth_basic_user_file /etc/nginx/conf.d/registry.password;
# add_header 'Docker-Distribution-Api-Version' 'registry/2.0' always;
```

Uncomment the two lines that start with `auth_basic` as well as the line that starts with `add_header` by removing the # character at the beginning of the lines. It should then look like this:

~/docker-registry/nginx/registry.conf

```
# To add basic authentication to v2 use auth_basic setting plus add_header
auth_basic "registry.localhost";
auth_basic_user_file /etc/nginx/conf.d/registry.password;
add_header 'Docker-Distribution-Api-Version' 'registry/2.0' always;
```

We've now told Nginx to enable HTTP basic authentication for all requests that get proxied to the Docker registry and told it to use the password file we just created.

Let's bring our containers back up to see if authentication is working:

```
cd ~/docker-registry
docker-compose up
```

Repeat the previous curl test:

```
curl http://localhost:5043/v2/
```

You should get a message complaining about being unauthorized:

```
Output of curl
<html>
<head><title>401 Authorization Required</title></head>
<body bgcolor="white">
<center><h1>401 Authorization Required</h1></center>
<hr><center>nginx/1.9.7</center>
</body>
</html>
```

Now try adding the username and password you created earlier to the `curl` request:

```
curl http://USERNAME:PASSWORD@localhost:5043/v2/
```

You should get the same output you were getting before — the empty json object `{}`. You should also see the same `registry_` output in the `docker-compose` terminal.

Go ahead and use `CTRL-C` in the `docker-compose` terminal to shut down the Docker containers.

Step 5 — Setting Up SSL

At this point we have the registry up and running behind Nginx with HTTP basic authentication working. However, the setup is still not very secure since the connections are unencrypted. You might have noticed the commented-out SSL lines in the Nginx config file we made earlier.

Let's enable them. First, open the Nginx configuration file for editing:

```
nano ~/docker-registry/nginx/registry.conf
```

Use the arrow keys to move around and look for these lines:

~/docker-registry/nginx/registry.conf

```
server {
  listen 443;
  server_name myregistrydomain.com;

  # SSL
  # ssl on;
  # ssl_certificate /etc/nginx/conf.d/domain.crt;
  # ssl_certificate_key /etc/nginx/conf.d/domain.key;
```

Uncomment the lines below the SSL comment by removing the # characters in front of them. If you have a domain name set up for your server, change the value of `server_name` to your domain name while you're at it. When you're done, the top of the file should look like this:

~/docker-registry/nginx/registry.conf

```
server {
  listen 443;
  server_name myregistrydomain.com;

  # SSL
  ssl on;
  ssl_certificate /etc/nginx/conf.d/domain.crt;
  ssl_certificate_key /etc/nginx/conf.d/domain.key;
```

Save the file. Nginx is now configured to use SSL and will look for the SSL certificate and key files at `/etc/nginx/conf.d/domain.crt` and `/etc/nginx/conf.d/domain.key` respectively. Due to the mappings we set up earlier in our `docker-compose.yml` file the `/etc/nginx/conf.d/` path in the

Nginx container corresponds to the folder `~/docker-registry/nginx/` on our host machine, so we'll put our certificate files there.

If you already have an SSL certificate set up or are planning to buy one, then you can just copy the certificate and key files to the paths listed in `registry.conf` (`ssl_certificate` and `ssl_certificate_key`).

You could also get a free signed SSL certificate.

Otherwise we'll have to use a self-signed SSL certificate.

Signing Your Own Certificate

Since Docker currently doesn't allow you to use self-signed SSL certificates this is a bit more complicated than usual — we'll also have to set up our system to act as our own certificate signing authority.

To begin, let's change to our `~/docker-registry/nginx` folder and get ready to create the certificates:

```
cd ~/docker-registry/nginx
```

Generate a new root key:

```
openssl genrsa -out devdockerCA.key 2048
```

Generate a root certificate (enter whatever you'd like at the prompts):

```
openssl req -x509 -new -nodes -key devdockerCA.key -days 10000 -out devdockerCA.crt
```

Then generate a key for your server (this is the file referenced by `ssl_certificate_key` in our Nginx configuration):

```
openssl genrsa -out domain.key 2048
```

Now we have to make a certificate signing request.

After you type this command, OpenSSL will prompt you to answer a few questions. Write whatever you'd like for the first few, but when OpenSSL prompts you to enter the **"Common Name" make sure to type in the domain or IP of your server**.

```
openssl req -new -key domain.key -out dev-docker-registry.com.csr
```

For example, if your Docker registry is going to be running on the domain **`www.ilovedocker.com`**, then your input should look like this:

```
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
```

```
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:www.ilovedocker.com
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Do not enter a challenge password.

Next, we need to sign the certificate request:

```
openssl x509 -req -in dev-docker-registry.com.csr -CA devdockerCA.crt -CAkey devdockerCA.key -CAcreateserial -out domain.crt -days 10000
```

Since the certificates we just generated aren't verified by any known certificate authority (e.g., VeriSign), we need to tell any clients that are going to be using this Docker registry that this is a legitimate certificate. Let's do this locally on the host machine so that we can use Docker from the Docker registry server itself:

```
sudo mkdir /usr/local/share/ca-certificates/docker-dev-cert
sudo cp devdockerCA.crt /usr/local/share/ca-certificates/docker-dev-cert
sudo update-ca-certificates
```

Restart the Docker daemon so that it picks up the changes to our certificate store:

```
sudo service docker restart
```

Warning: You'll have to repeat this step for every machine that connects to this Docker registry! Instructions for how to do this for Ubuntu 14.04 clients are listed in *Step 9 — Accessing Your Docker Registry from a Client Machine*.

Step 6 — Testing SSL

Bring up our Docker containers via the now familiar `docker-compose up`:

```
cd ~/docker-registry
docker-compose up
```

Do another `curl` test from another terminal (only this time using https) to verify that our SSL setup is working properly. Keep in mind that for SSL to work correctly you will have to use the same domain name you typed into the `Common Name` field earlier while you were creating your SSL certificate.

```
curl https://USERNAME:PASSWORD@[YOUR-DOMAIN]:5043/v2/
```

Note: If you are using a self-signed certificate, you will see the following error from `curl`:

```
curl: (60) SSL certificate problem: self signed certificate
```

Use the `-k` option to tell `curl` *not* to verify with the peer:

```
curl -k https://USERNAME:PASSWORD@[YOUR-DOMAIN]:5043/v2/
```

For example, if the user and password you set up were `sammy` and `test`, and your SSL certificate is for `www.example.com`, then you would type the following:

```
curl https://sammy:test@www.example.com:5043/v2/
```

If all went well `curl` will print an empty json object `{}`, and your `docker-compose` terminal will print the usual `registry_` output.

If not, recheck the SSL steps and your Nginx configuration file to make sure everything is correct.

At this point we have a functional Docker registry 2.0 up and running behind an Nginx server which is providing authentication and encryption via SSL. If your firewall is configured to allow access to port `5043` from the outside, then you should be able to login to this Docker registry from any machine `docker login https://<YOURDOMAIN>` and entering the username and password you set in the earlier section.

Step 7 — Setting SSL Port to 443

Just a couple more steps to do before we're done: change the port to use the standard SSL port of `443` (optional) and set `docker-compose` up to start this set of containers on startup.

Let's start by setting up our dockerized Nginx container to listen on port 443 (the standard port for SSL) rather than the non-standard port 5043 we've been using so far. Ports below 1024 are "privileged" ports on Linux though, which means we're going to have to run our `docker-compose` container as root.

First open up `docker-compose.yml` in a text editor:

```
nano ~/docker-registry/docker-compose.yml
```

Under the Nginx section you'll see a `ports:` section, change the `- 5043:443` line (this maps port 5043 on our host machine to port 443 inside the Nginx container) to `- 443:443` so that our Nginx container's port 443 gets mapped to our host machine's port 443. When finished your `docker-compose.yml` should look like this:

~/docker-registry/docker-compose.yml

```
nginx:
  image: "nginx:1.9"
  ports:
    - 443:443
  links:
    - registry:registry
  volumes:
    - ./nginx/:/etc/nginx/conf.d:ro
registry:
  image: registry:2
  ports:
    - 127.0.0.1:5000:5000
  environment:
    REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
  volumes:
    - ./data:/data
```

Kill your `docker-compose` session via **CTRL-C** if it's still running, and restart it on port 443:

```
sudo docker-compose up
```

Note: Only root users can listen to ports below 1024. Notice that you need to use `sudo` this time with the `docker-compose` command so Nginx can run on the default SSL port 443.

You should see `docker-compose` start up as usual.

Let's try another `curl` test using our domain name, only this time we won't specify the `:5043` in the URL:

```
curl https://<YOURUSERNAME>:<YOURPASSWORD>@YOUR-DOMAIN/v2/
```

If all went well you should see the usual `registry_` output in your `docker-compose` terminal. You may also want to try running this same `curl` command from another machine to make sure that your port 443 is being exposed to the outside world.

Go ahead and use `CTRL-C` in the `docker-compose` terminal to shut down the Docker containers before moving to the next step.

Step 8 — Starting Docker Registry as a Service

If everything is looking good, let's go ahead and create an Upstart script so that our Docker registry will start whenever the system boots up.

First let's remove any existing containers, move our Docker registry to a system-wide location and change its

permissions to root:

```
cd ~/docker-registry
docker-compose rm   # this removes the existing containers
sudo mv ~/docker-registry /docker-registry
sudo chown -R root: /docker-registry
```

Then use your favorite text editor to create an Upstart script:

```
sudo nano /etc/init/docker-registry.conf
```

Add the following contents to create the Upstart script (getting Upstart to properly monitor Docker containers is a bit tricky, check out this blog post if you'd like more info about what this Upstart script is doing):

/etc/init/docker-registry.conf

```
description "Docker Registry"

start on runlevel [2345]
stop on runlevel [016]

respawn
respawn limit 10 5

chdir /docker-registry

exec /usr/local/bin/docker-compose up
```

For more about Upstart scripts, please read this tutorial.

Let's test our new Upstart script by running:

```
sudo service docker-registry start
```

You should see something like this:

```
docker-registry start/running, process 25303
```

You can verify that the server is running by executing:

```
docker ps
```

The output should look similar to the following (note that the names all start with `dockerregistry_`

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                           NAMES
d4b6fef0b4d1        nginx:1.9           "nginx -g 'daemon of   2 minutes ago       Up 2
```

```
minutes        80/tcp, 0.0.0.0:443->443/tcp   dockerregistry_nginx_1
77668352bd39         registry:2           "registry cmd/regist   2 minutes ago        Up 2
minutes        127.0.0.1:5000->5000/tcp       dockerregistry_registry_1
```

Upstart will log the output of the `docker-compose` command to `/var/log/upstart/docker-registry.log`. For our final test let's "live-watch" the log file with `tail` (the `sudo` is necessary because upstart logs are written as the root user):

```
sudo tail -f /var/log/upstart/docker-registry.log
```

You should see the usual `registry_` output. From another terminal or machine go ahead and run our now familiar `curl` test:

```
curl https://<YOUR_USERNAME>:<YOURPASSWORD>@[YOUR-DOMAIN]/v2/
```

If everything is working correctly curl will print a `{}` to your terminal and you should see the usual:

```
registry_1  |  time="2015-08-12T08:01:12.241887501Z"  level=debug  msg="authorizing  request"
environment=development                        http.request.host=docker.meatflavoredbeer.com
http.request.id=e8d69e16-9448-4c48-afd8-57b1f1302742                http.request.method=GET
http.request.remoteaddr=106.1.247.4                                 http.request.uri="/v2/"
http.request.useragent="curl/7.37.1"       instance.id=14d4727b-fda1-463f-8d0e-181f4c70cb17
service=registry version=v2.0.1
registry_1  |  time="2015-08-12T08:01:12.242206499Z"  level=info  msg="response  completed"
environment=development                        http.request.host=docker.meatflavoredbeer.com
http.request.id=e8d69e16-9448-4c48-afd8-57b1f1302742                http.request.method=GET
http.request.remoteaddr=106.1.247.4                                 http.request.uri="/v2/"
http.request.useragent="curl/7.37.1"           http.response.contenttype="application/json;
charset=utf-8"          http.response.duration=3.359883ms          http.response.status=200
http.response.written=2  instance.id=14d4727b-fda1-463f-8d0e-181f4c70cb17  service=registry
version=v2.0.1
registry_1 | 172.17.0.4 - - [12/Aug/2015:08:01:12 +0000] "GET /v2/ HTTP/1.0" 200 2 ""
"curl/7.37.1"
nginx_1    | 106.1.247.4 - nik [12/Aug/2015:08:01:12 +0000] "GET /v2/ HTTP/1.1" 200 2 "-"
"curl/7.37.1" "-"
```

Step 9 — Accessing Your Docker Registry from a Client Machine

To access your Docker registry from another machine, first add the SSL certificate you created earlier to the new client machine. The file you want is located at `~/docker-registry/nginx/devdockerCA.crt`.

You can copy it to the new machine directly or use the below instructions to copy and paste it:

On the **registry server**, view the certificate:

```
sudo cat /docker-registry/nginx/devdockerCA.crt
```

You'll get output that looks something like this:

```
Output of sudo cat /docker-registry/nginx/devdockerCA.crt
-----BEGIN CERTIFICATE-----
MIIDXTCCAkWgAwIBAgIJANiXy7fHSPrmMA0GCSqGSIb3DQEBCwUAMEUxCzAJBgNV
BAYTAkFVMRMwEQYDVQQIDApTb21lLVN0YXRlMSEwHwYDVQQKDBhJbnRlcm5ldCBX
aWRnaXRzIFB0eSBMdGQwHhcNMTQwOTIxMDYwODE2WhcNNDIwMjA2MDYwODE2WjBF
MQswCQYDVQQGEwJBVTETMBEGA1UECAwKU29tZS1TdGF0ZTEhMB8GA1UECgwYSW50
ZXJuZXQgV2lkZ2l0cyBQdHkgTHRkMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIB
CgKCAQEAuK4kNFaY3k/0RdKRK1XLj9+IrpR7WW5lrNaFB0OIiItHV9FjyuSWK2mj
ObR1IWJNrVSqWvfZ/CLGay6Lp9DJvBbpT68dhuS5xbVw3bs3ghB24TntDYhHMAc8
GWor/ZQTzjccHUd1SJxt5mGXalNHUharkLd8mv4fAb7Mh/7AFP32W4X+scPE2bVH
OJ1qH8ACo7pSVl1Ohcri6sMp01GoELyykpXu5azhuCnfXLRyuOvQb7llV5WyKhq+
SjcE3c2C+hCCC5g6IzRcMEg336Ktn5su+kK6c0hoD0PR/W0PtwgH4XlNdpVFqMST
vthEG+Hv6xVGGH+nTszN7F9ugVMxewIDAQABo1AwTjAdBgNVHQ4EFgQULek+WVyK
dJk3JIHoI4iVi0FPtdwwHwYDVR0jBBgwFoAULek+WVyKdJk3JIHoI4iVi0FPtdww
DAYDVR0TBAUwAwEB/zANBgkqhkiG9w0BAQsFAAOCAQEAkignESZcgr4dBmVZqDwh
YsrKeWSkj+5p9eW5hCHJ5Eg2X8oGTgItuLaLfyFWPS3MYWWMzggxgKMOQM+9o3+k
oH5sUmraNzI3TmAtkqd/8isXzBUV661BbSV0obAgF/ul5v3Tl5uBbCXObC+NUikM
O0C3fDmmeK799AM/hP5CTDehNaFXABGoVRMSlGYe8hZqap/Jm6AaKThV4g6n4F7M
u5wYtI9YDMsxeVW6OP9ZfvpGZW/n/88MSFjMlBjFfFsorfRd6P5WADhdfA6CBECG
LP83r7/MhqO06EOpsv4n2CJ3yoyqIr1L1+6C7Erl2em/jfOb/24y63dj/ATytt2H
6g==
-----END CERTIFICATE-----
```

Copy that output to your clipboard, and connect to your client machine.

On the **client machine**, create the certificate directory:

```
sudo mkdir /usr/local/share/ca-certificates/docker-dev-cert
```

Open the certificate file for editing:

```
sudo nano /usr/local/share/ca-certificates/docker-dev-cert/devdockerCA.crt
```

Paste the certificate contents.

Verify that the file saved to the client machine correctly by viewing the file:

```
cat /usr/local/share/ca-certificates/docker-dev-cert/devdockerCA.crt
```

If everything worked properly you'll see the same text from earlier:

```
Output of cat /usr/local/share/ca-certificates/docker-dev-cert/devdockerCA.crt
-----BEGIN CERTIFICATE-----
MIIDXTCCAkWgAwIBAgIJANiXy7fHSPrmMA0GCSqGSIb3DQEBCwUAMEUxCzAJBgNV
...
...
LP83r7/MhqO06EOpsv4n2CJ3yoyqIr1L1+6C7Erl2em/jfOb/24y63dj/ATytt2H
6g==
-----END CERTIFICATE-----
```

Now update the certificates:

```
sudo update-ca-certificates
```

You should get output that looks like the following (note the`1 added`):

```
Output of sudo update-ca-certificates
Updating certificates in /etc/ssl/certs... 1 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d....done.
```

If you don't have Docker installed on the client yet, do so now (see the Prerequisites section).

Restart Docker to make sure it reloads the system's CA certificates.

```
sudo service docker restart
```

You should now be able to log in to your Docker registry from the client machine:

```
docker login https://YOUR-DOMAIN
```

Note that you're using **`https://`**. Enter the username and password you set up earlier (enter whatever you'd like for email if prompted).

```
Output of docker login
Username: USERNAME
Password: PASSWORD
Email:
Account created. Please see the documentation of the registry http://localhost:5000/v1/ for instructions how to activate it.
```

You should see the following message:

```
Output of docker login
Login Succeeded
```

At this point your Docker registry is up and running! Let's make a test image to push to the registry.

Step 10 — Publish to Your Private Docker Registry

You are now ready to publish an image to your private Docker registry, but first we have to create an image. We will create a simple image based on the `ubuntu` image from Docker Hub.

From your **client machine**, create a small empty image to push to our new registry.

```
docker run -t -i ubuntu /bin/bash
```

After it finishes downloading you'll be inside a Docker prompt. Let's make a quick change to the filesystem by creating a file called `SUCCESS`:

```
touch /SUCCESS
```

Exit out of the Docker container:

```
exit
```

Commit the change:

```
docker commit $(docker ps -lq) test-image
```

This command creates a new image called `test-image` based on the image already running plus any changes you have made. In our case, the addition of the `/SUCCESS` file is included in the new image.

This image only exists locally right now, so let's push it to the new registry we've created.

In the previous step, you logged into your private Docker registry. In case you aren't still logged in, let's log in again (note that you want to use **`https://`**):

```
docker login https://YOUR-DOMAIN
```

Enter the username and password you set up earlier:

```
Username: USERNAME
Password: PASSWORD
Email:
Account created. Please see the documentation of the registry http://localhost:5000/v1/ for instructions how to activate it.
```

Docker has an unusual mechanism for specifying which registry to push to. You have to tag an image with the private registry's location in order to push to it. Let's tag our image to our private registry:

```
docker tag test-image [YOUR-DOMAIN]/test-image
```

Note that you are using the local name of the image first, then the tag you want to add to it. The tag does **not**

use `https://`, just the domain, port, and image name.

Now we can push that image to our registry. This time we're using the tag name only:

```
docker push [YOUR-DOMAIN]/test-image
```

This will take a moment to upload to the registry server. You should see output that ends with something similar to the following:

```
Output of docker push
latest: digest: sha256:5ea1cfb425544011a3198757f9c6b283fa209a928caabe56063f85f3402363b4
size: 8008
```

Step 11 — Pull from Your Docker Registry

To make sure everything worked, let's go back to our original server (where you installed the Docker registry) and pull the image we just pushed from the client. You could also test this from a third server.

If Docker is not installed on your test pull server, go back and follow the installation instructions (and if it's a third server, the SSL instructions) from Step 6.

Log in with the username and password you set up previously.

```
docker login https://[YOUR-DOMAIN]
```

And now pull the image. You want just the "tag" image name, which includes the domain name, port, and image name (but not `https://`):

```
docker pull [YOUR-DOMAIN]/test-image
```

Docker will do some downloading and return you to the prompt. If you run the image on the new machine you'll see that the SUCCESS file we created earlier is there:

```
docker run -t -i [YOUR-DOMAIN]/test-image /bin/bash
```

List your files inside the bash shell:

```
ls
```

You should see the SUCCESS file we created earlier for this image:

```
SUCCESS bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv
sys tmp usr var
```

```
docker pull and docker push worked fine....but docker search didn't
```

```
so use the following for see the images
```

ssh bizruntime@biz.com tree /docker-registry/data/docker/registry/v2/repositories -L 2

Conclusion

Congratulations! You've just used your own private Docker registry to push and pull your first Docker container!