How To Set Up a Private Docker Registry on Ubuntu 18.04

Posted January 7, 2019 3.7k DOCKER NGINX UBUNTU 18.04

5

By: Young Kim

The author selected the Apache Software Foundation to receive a donation as part of the Write for DOnations program.

Introduction

Docker Registry is an application that manages storing and delivering Docker container images. Registries centralize container images and reduce build times for developers. Docker images guarantee the same runtime environment through virtualization, but building an image can involve a significant time investment. For example, rather than installing dependencies and packages separately to use Docker, developers can download a compressed image from a registry that contains all of the necessary components. Furthermore, developers can automate pushing images to a registry using continuous integration tools, such as TravisCI, to seamlessly update images during production and development.

Docker also has a free public registry, Docker Hub, that can host your custom Docker images, but there are situations where you will not want your image to be publicly available. Images typically contain all the code necessary to run an application, so using a private registry is preferable when using proprietary software.

In this tutorial, you will set up and secure your own private Docker Registry. You will use Docker Compose to define configurations to run your Docker applications and Nginx to forward server traffic from HTTPS to the running Docker container. Once you've completed this tutorial, you will be able to push a custom Docker image to your private registry and pull the image securely from a remote server.

Prerequisites

Before you begin this guide, you'll need the following:

- Two Ubuntu 18.04 servers set up by following the Ubuntu 18.04 initial server setup guide, including a sudo non-root user and a firewall. One server will host your private Docker Registry and the other will be your **client** server.
- Docker and Docker-Compose installed on both servers by following the How to Install Docker-Compose on Ubuntu 18.04 tutorial. You only need to complete the first step of this tutorial to install Docker Compose. This tutorial explains how to install Docker as part of its prerequisites.

- Nginx installed on your private Docker Registry server by following the How to Install Nginx on Ubuntu 18.04.
- Nginx secured with Let's Encrypt on your server for the private Docker Registry, by following How to Secure Nginx With Let's Encrypt. Make sure to redirect all traffic from HTTP to HTTPS in Step 4.
- A domain name that resolves to the server you're using for the private Docker Registry. You will set this up as part of the Let's Encrypt prerequisite.

Step 1 — Installing and Configuring the Docker Registry

The Docker command line tool is useful for starting and managing one or two Docker containers, but, for full deployment most applications running inside Docker containers require other components to be running in parallel. For example, a lot of web applications consist of a web server, like Nginx, that serves up the application's code, an interpreted scripting language such as PHP, and a database server like MySQL.

With Docker Compose, you can write one `.yml` file to set up each container's configuration and the information the containers need to communicate with each other. You can then use the `docker-compose` command line tool to issue commands to all the components that make up your application.

Docker Registry is itself an application with multiple components, so you will use Docker Compose to manage your configuration. To start an instance of the registry, you'll set up a `docker-compose.yml` file to define the location where your registry will be storing its data.

On the server you have created to host your private Docker Registry, you can create a `docker-registry` directory, move into it, and then create a `data` subfolder with the following commands:

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

Use your text editor to create the `docker-compose.yml` configuration file:

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

Add the following content to the file, which describes the basic configuration for a Docker Registry:

docker-compose.yml

```
version: '3'

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

The `environment` section sets an environment variable in the Docker Registry container with the path `/data`. The Docker Registry application checks this environment variable when it starts up, and as a result begins to save its data to the `/data` folder.

However, as you have included the `volumes: - ./data:/data` line, Docker will start to map the `/data` directory in that container to `/data` on your registry server. The end result is that the Docker Registry's data all gets stored in `~/docker-registry/data` on the registry server.

The `ports` section, with configuration `5000:5000`, tells Docker to map port `5000` on the server to port `5000` in the running container. This allows you to send a request to port `5000` on the server, and have the request forwarded to the registry application.

You can now start Docker Compose to check the setup:

```
$ docker-compose up
```

You will see download bars in your output that show Docker downloading the Docker Registry image from Docker's own registry. Within a minute or two, you'll see output that looks similar to the following (versions might vary):

```
Output of docker-compose up
Starting docker-registry_registry_1 ... done
Attaching to docker-registry_registry_1
registry_1  | time="2018-11-06T18:43:09Z" level=warning msg="No HTTP secret provided - generated rand
registry_1  | time="2018-11-06T18:43:09Z" level=info msg="redis not configured" go.version=go1.7.6 in
registry_1  | time="2018-11-06T18:43:09Z" level=info msg="Starting upload purge in 20m0s" go.version=
registry_1  | time="2018-11-06T18:43:09Z" level=info msg="using inmemory blob descriptor cache" go.ve
registry_1  | time="2018-11-06T18:43:09Z" level=info msg="listening on [::]:5000" go.version=go1.7.6
```

You'll address the `No HTTP secret provided` warning message later in this tutorial. The output shows that the container is starting. The last line of the output shows it has successfully started listening on port `5000`.

By default, Docker Compose will remain waiting for your input, so hit `CTRL+C` to shut down your Docker Registry container.

You have set up a full Docker Registry listening on port `5000`. At this point the registry won't start unless you bring it up manually. Also, Docker Registry doesn't come with any built-in authentication mechanism, so it is currently insecure and completely open to the public. In the following steps, you will address these security concerns.

Step 2 — Setting Up Nginx Port Forwarding

You already have HTTPS set up on your Docker Registry server with Nginx, which means you can now set up port forwarding from Nginx to port `5000`. Once you complete this step, you can access your registry directly at example.com.

As part of the How to Secure Nginx With Let's Encrypt prerequisite, you have already set up the `/etc/nginx/sites-available/example.com` file containing your server configuration.

Open this file with your text editor:

```
$ sudo nano /etc/nginx/sites-available/example.com
```

Find the existing `location` line. It will look like this:

/etc/nginx/sites-available/example.com

```
...
location / {
  ...
}
...
```

You need to forward traffic to port `5000`, where your registry will be running. You also want to append headers to the request to the registry, which provide additional information from the server with each request and response. Delete the contents of the `location` section, and add the following content into that section:

/etc/nginx/sites-available/example.com

```
...
location / {
    # 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;
    }

    proxy_pass                          http://localhost:5000;
    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;
}
...
```

The `$http_user_agent` section verifies that the Docker version of the client is above `1.5`, and ensures that the `UserAgent` is not a `Go` application. Since you are using version `2.0` of the registry, older clients are not supported. For more information, you can find the `nginx` header configuration in Docker's Registry Nginx guide.

Save and exit the file. Apply the changes by restarting Nginx:

```
$ sudo service nginx restart
```

You can confirm that Nginx is forwarding traffic to port `5000` by running the registry:

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

In a browser window, open up the following url:

```
https://example.com/v2
```

You will see an empty JSON object, or:

```
{}
```

In your terminal, you'll see output similar to the following:

```
Output of docker-compose up
registry_1  | time="2018-11-07T17:57:42Z" level=info msg="response completed" go.version=go1.7.6 http
registry_1  | 172.18.0.1 - - [07/Nov/2018:17:57:42 +0000] "GET /v2/ HTTP/1.0" 200 2 "" "Mozilla/5.0 (
```

You can see from the last line that a `GET` request was made to `/v2/`, which is the endpoint you sent a request to from your browser. The container received the request you made, from the port forwarding, and returned a response of `{}`. The code `200` in the last line of the output means that the container handled the request successfully.

Now that you have set up port forwarding, you can move on to improving the security of your registry.

Step 3 — Setting Up Authentication

With Nginx proxying requests properly, you can now secure your registry with HTTP authentication to manage who has access to your Docker Registry. To achieve this, you'll create an authentication file with `htpasswd` and add users to it. HTTP authentication is quick to set up and secure over a HTTPS connection, which is what the registry will use.

You can install the `htpasswd` package by running the following:

```
$ sudo apt install apache2-utils
```

Now you'll create the directory where you'll store our authentication credentials, and change into that directory. `$_` expands to the last argument of the previous command, in this case `~/docker-registry/auth`:

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

Next, you will create the first user as follows, replacing `username` with the username you want to use. The `-B` flag specifies `bcrypt` encryption, which is more secure than the default encryption. Enter the password when prompted:

```
$ htpasswd -Bc registry.password username
```

Note: To add more users, re-run the previous command without the -c option, (the `c` is for create):

```
  $ htpasswd registry.password username
```

Next, you'll edit the `docker-compose.yml` file to tell Docker to use the file you created to authenticate users.

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

You can add environment variables and a volume for the `auth/` directory that you created, by editing the `docker-compose.yml` file to tell Docker how you want to authenticate users. Add the following highlighted content to the file:

docker-compose.yml

```
version: '3'

services:
  registry:
    image: registry:2
    ports:
    - "5000:5000"
    environment:
      REGISTRY_AUTH: htpasswd
      REGISTRY_AUTH_HTPASSWD_REALM: Registry
      REGISTRY_AUTH_HTPASSWD_PATH: /auth/registry.password
      REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
    volumes:
      - ./auth:/auth
      - ./data:/data
```

For `REGISTRY_AUTH`, you have specified `htpasswd`, which is the authentication scheme you are using, and set `REGISTRY_AUTH_HTPASSWD_PATH` to the path of the authentication file. Finally, `REGISTRY_AUTH_HTPASSWD_REALM` signifies the name of `htpasswd` realm.

You can now verify that your authentication works correctly, by running the registry and checking that it prompts users for a username and password.

```
$ docker-compose up
```

In a browser window, open `https://example.com/v2`.

After entering `username` and the corresponding password, you will see `{}` once again. You've confirmed the basic authentication setup: the registry only returned the result after you entered the correct username and password. You have now secured your registry, and can continue to using the registry.

Step 4 — Starting Docker Registry as a Service

You want to ensure that your registry will start whenever the system boots up. If there are any unforeseen system crashes, you want to make sure the registry restarts when the server reboots. Open up `docker-compose.yml`:

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

Add the following line of content under `registry:`:

docker-compose.yml

```
...
  registry:
    restart: always
...
```

You can start your registry as a background process, which will allow you to exit the `ssh` session and persist the process:

```
$ docker-compose up -d
```

With your registry running in the background, you can now prepare Nginx for file uploads.

Step 5 — Increasing File Upload Size for Nginx

Before you can push an image to the registry, you need to ensure that your registry will be able to handle large file uploads. Although Docker splits large image uploads into separate layers, they can sometimes be over `1GB`. By default, Nginx has a limit of `1MB` on file uploads, so you need to edit the configuration file for `nginx` and set the max file upload size to `2GB`.

```
$ sudo nano /etc/nginx/nginx.conf
```

Find the `http` section, and add the following line:

/etc/nginx/nginx.conf

```
...
http {
        client_max_body_size 2000M;
        ...
}
...
```

Finally, restart Nginx to apply the configuration changes:

```
$ sudo service nginx restart
```

You can now upload large images to your Docker Registry without Nginx errors.

Step 6 — Publishing to Your Private Docker Registry

You are now ready to publish an image to your private Docker Registry, but first you have to create an image. For this tutorial, you will create a simple image based on the `ubuntu` image from Docker Hub. Docker Hub is a publicly hosted registry, with many pre-configured images that can be leveraged to quickly Dockerize applications. Using the `ubuntu` image, you will test pushing and pulling to your registry.

From your **client** server, create a small, empty image to push to your new registry, the `-i` and `-t` flags give you interactive shell access into the container:

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

After it finishes downloading you'll be inside a Docker prompt, note that your container ID following `root@` will vary. Make a quick change to the filesystem by creating a file called `SUCCESS`. In the next step, you'll be able to use this file to determine whether the publishing process is successful:

```
root@f7e13d5464d1:/# touch /SUCCESS
```

Exit out of the Docker container:

```
root@f7e13d5464d1:/# exit
```

The following 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.

Commit the change:

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

At this point, the image only exists locally. Now you can push it to the new registry you have created. Log in to your Docker Registry:

```
$ docker login https://example.com
```

Enter the `username` and corresponding password from earlier. Next, you will tag the image with the private registry's location in order to push to it:

```
$ docker tag test-image example.com/test-image
```

Push the newly tagged image to the registry:

```
$ docker push example.com/test-image
```

Your output will look similar to the following:

```
Output
The push refers to a repository [example.com/test-image]
e3fbbfb44187: Pushed
5f70bf18a086: Pushed
a3b5c80a4eba: Pushed
7f18b442972b: Pushed
3ce512daaf78: Pushed
7aae4540b42d: Pushed
...
```

You've verified that your registry handles user authentication, and allows authenticated users to push images to the registry. Next, you will confirm that you are able to pull images from the registry as well.

Step 7 — Pulling From Your Private Docker Registry

Return to your registry server so that you can test pulling the image from your **client** server. It is also possible to test this from a third server.

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

```
$ docker login https://example.com
```

You're now ready to pull the image. Use your domain name and image name, which you tagged in the previous step:

```
$ docker pull example.com/test-image
```

Docker will download the image and return you to the prompt. If you run the image on the registry server you'll see the `SUCCESS` file you created earlier is there:

```
$ docker run -it example.com/test-image /bin/bash
```

List your files inside the bash shell:

```
$ ls
```

You will see the `SUCCESS` file you created for this image:

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

You've finished setting up a secure registry to which users can push and pull custom images.

Conclusion

In this tutorial you set up your own private Docker Registry, and published a Docker image. As mentioned in the introduction, you can also use TravisCI or a similar CI tool to automate pushing to a private registry directly. By leveraging Docker and registries into your workflow, you can ensure that the image containing the code will result in the same behavior on any machine, whether in production or in development. For more information on writing Docker files, you can read this Docker tutorial explaining the process.

By: Young Kim

Upvote (5)   Subscribe   Share

Editor:
Kathryn Hancox

Related Tutorials

How To Install Docker Compose on Ubuntu 18.04

How To Remove Docker Images, Containers, and Volumes

Naming Docker Containers: 3 Tips for Beginners

How to Manually Set Up a Prisma Server on Ubuntu 18.04

How To Use Traefik as a Reverse Proxy for Docker Containers on Debian 9

0 Comments

Leave a comment...

Log In to Comment