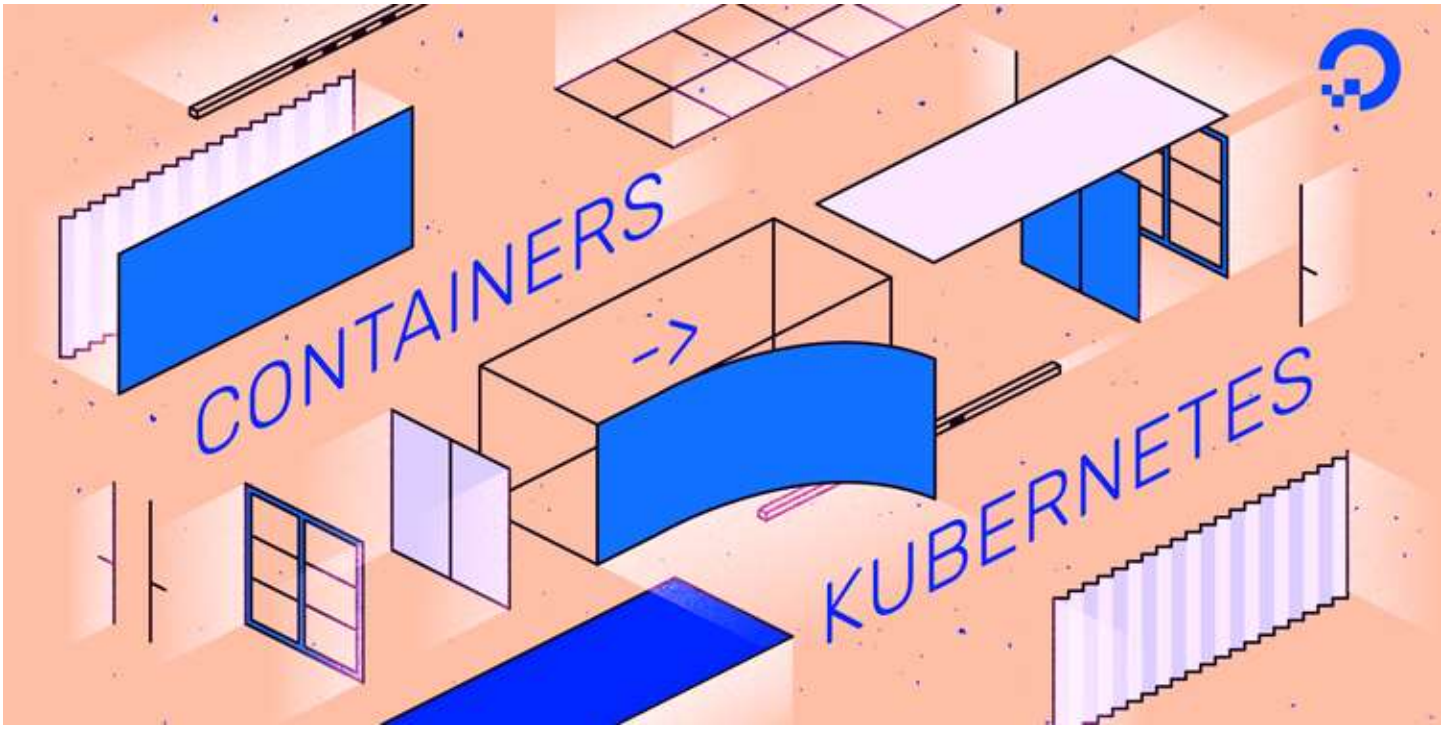


TUTORIAL

How To Secure a Containerized Node.js Application with Nginx, Let's Encrypt, and Docker Compose

Node.js Security Docker Let's Encrypt Ubuntu 18.04

By Kathleen Juell
Published on January 4, 2019 192.6k

English

Introduction

Implementing a reverse proxy with TLS/SSL on containers involves a different set of procedures from working directly on a host operating system. For example, if you were

In this tutorial, you will deploy a Node.js application with an Nginx reverse proxy using Docker Compose. You will obtain TLS/SSL certificates for the domain associated with your application and ensure that it receives a high security rating from SSL Labs. Finally, you will set up a `cron` job to renew your certificates so that your domain remains secure.

Prerequisites

To follow this tutorial, you will need:

- An Ubuntu 18.04 server, a non-root user with `sudo` privileges, and an active firewall. For guidance on how to set these up, please see this Initial Server Setup guide.
- Docker and Docker Compose installed on your server. For guidance on installing Docker, follow Steps 1 and 2 of How To Install and Use Docker on Ubuntu 18.04. For guidance on installing Compose, follow Step 1 of How To Install Docker Compose on Ubuntu 18.04.
- A registered domain name. This tutorial will use **example.com** throughout. You can get one for free at Freenom, or use the domain registrar of your choice.
- Both of the following DNS records set up for your server. You can follow this introduction to DigitalOcean DNS for details on how to add them to a DigitalOcean account, if that's what you're using:
    - An A record with `example.com` pointing to your server's public IP address.
    - An A record with `www.example.com` pointing to your server's public IP address.

In your non-root user's home directory, clone the `nodejs-image-demo` repository from the DigitalOcean Community GitHub account. This repository includes the code from the

Change to the `node_project` directory:

```
$ cd  node_project
```

In this directory, there is a Dockerfile that contains instructions for building a Node application using the Docker `node:10` image and the contents of your current project directory. You can look at the contents of the Dockerfile by typing:

```
$ cat Dockerfile
```

Output

```
FROM node:10-alpine

RUN mkdir -p /home/node/app/node_modules && chown -R node:node /home/node/app

WORKDIR /home/node/app

COPY package*.json ./

USER node

RUN npm install

COPY --chown=node:node . .
```

`package.json` and `package-lock.json`, containing the project's listed dependencies, from the copy of the rest of the application code. Finally, the instructions specify that the

To test the application without SSL, you can build and tag the image using `docker build` and the `-t` flag. We will call the image `node-demo`, but you are free to name it something else:

```
$ docker build -t node-demo .
```

Once the build process is complete, you can list your images with `docker images`:

```
$ docker images
```

You will see the following output, confirming the application image build:

```
Output
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
node-demo           latest              23961524051d        7 seconds ago       73MB
node                10-alpine           8a752d5af4ce        3 weeks ago         70.7MB
```

Next, create the container with `docker run`. We will include three flags with this command:

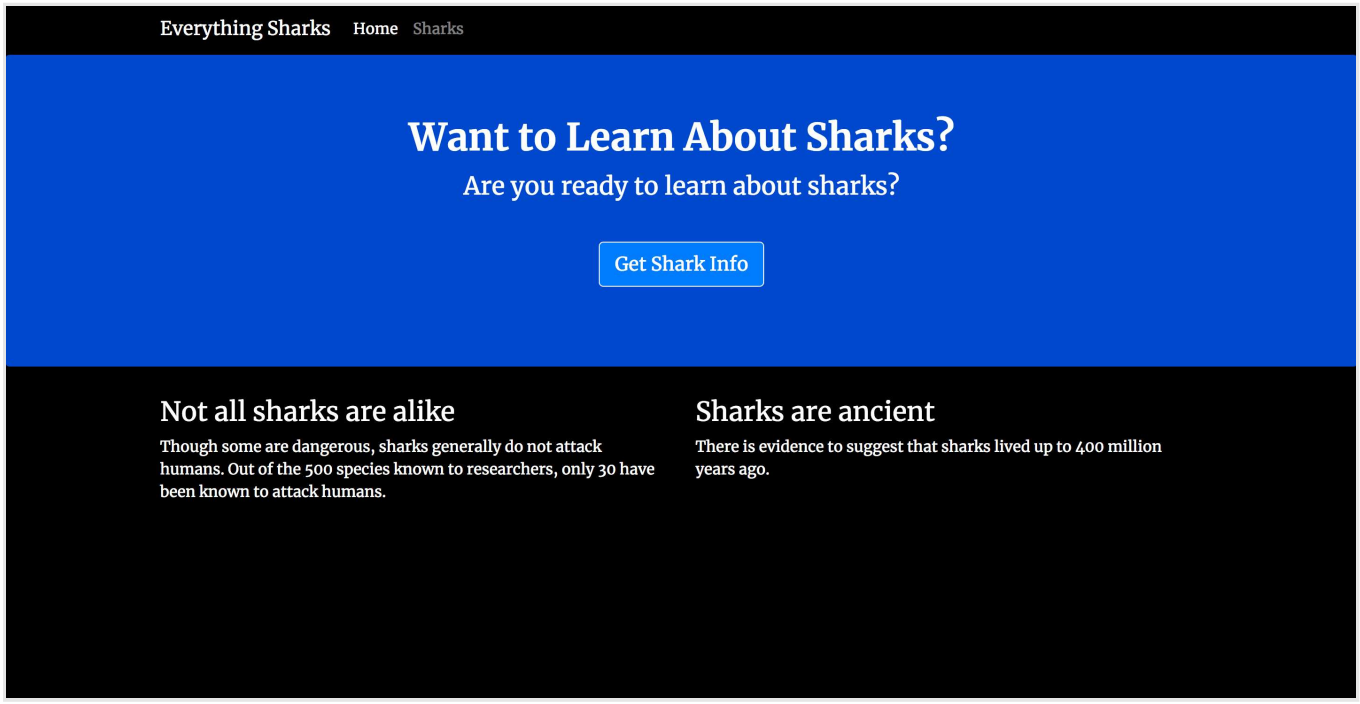
- `-p`: This publishes the port on the container and maps it to a port on our host. We will use port `80` on the host, but you should feel free to modify this as necessary if you have another process running on that port. For more information about how this works, see this discussion in the Docker docs on port binding.

```
$ docker run --name node-demo -p 80:8080 -d node-demo
```



```
Output
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
4133b72391da        node-demo           "node app.js"       17 seconds ago      Up 16 seco
```

You can now visit your domain to test your setup: `http://example.com` . Remember to replace `example.com` with your own domain name. Your application will display the following landing page:



Now that you have tested the application, you can stop the container and remove the

```
4133b72391da        node-demo           "node app.js"       17 seconds ago      Up 16 secc
```


You can now remove the stopped container and all of the images, including unused and dangling images, with `docker system prune` and the `-a` flag:

```
$ docker system prune -a
```

Type `y` when prompted in the output to confirm that you would like to remove the stopped container and images. Be advised that this will also remove your build cache.

With your application image tested, you can move on to building the rest of your setup with Docker Compose.

Step 2 — Defining the Web Server Configuration

With our application Dockerfile in place, we can create a configuration file to run our Nginx container. We will start with a minimal configuration that will include our domain name, document root, proxy information, and a location block to direct Certbot's requests to the `.well-known` directory, where it will place a temporary file to validate that the DNS for our domain resolves to our server.

First, create a directory in the current project directory for the configuration file:

```
$ mkdir nginx-conf
```

Open the file with `nano` or your favorite editor:

~/node_project/nginx-conf/nginx.conf


```
        location / {
                proxy_pass http://nodejs:8080;
        }

        location ~ /.well-known/acme-challenge {
                allow all;
                root /var/www/html;
        }
}
```

This server block will allow us to start the Nginx container as a reverse proxy, which will pass requests to our Node application container. It will also allow us to use Certbot's webroot plugin to obtain certificates for our domain. This plugin depends on the HTTP-01 validation method, which uses an HTTP request to prove that Certbot can access resources from a server that responds to a given domain name.

Once you have finished editing, save and close the file. To learn more about Nginx server and location block algorithms, please refer to this article on Understanding Nginx Server and Location Block Selection Algorithms.

With the web server configuration details in place, we can move on to creating our `docker-compose.yml` file, which will allow us to create our application services and the Certbot container we will use to obtain our certificates.

Step 3 — Creating the Docker Compose File

The `docker-compose.yml` file will define our services, including the Node application and

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



```
nodejs:
  build:
    context: .
    dockerfile: Dockerfile
  image: nodejs
  container_name: nodejs
  restart: unless-stopped
```

The `nodejs` service definition includes the following:

- `build`: This defines the configuration options, including the `context` and `dockerfile`, that will be applied when Compose builds the application image. If you wanted to use an existing image from a registry like Docker Hub, you could use the `image` instruction instead, with information about your username, repository, and image tag.
- `context`: This defines the build context for the application image build. In this case, it's the current project directory.
- `dockerfile`: This specifies the Dockerfile that Compose will use for the build — the Dockerfile you looked at in Step 1.
- `image`, `container_name`: These apply names to the image and container.
- `restart`: This defines the restart policy. The default is `no`, but we have set the container to restart unless it is stopped.

Note that we are not including bind mounts with this service, since our setup is focused on deployment rather than development. For more information, please see the Docker documentation on bind mounts and volumes.

```
  services:
    nodejs:
```


while exposing no ports to the outside world. Thus, you can be selective about opening only the ports you need to expose your frontend services.

Next, define the `webserver` service:

~/node_project/docker-compose.yml

```
...
 webserver:
    image: nginx:mainline-alpine
    container_name: webserver
    restart: unless-stopped
    ports:
      - "80:80"
    volumes:
      - web-root:/var/www/html
      - ./nginx-conf:/etc/nginx/conf.d
      - certbot-etc:/etc/letsencrypt
      - certbot-var:/var/lib/letsencrypt
    depends_on:
      - nodejs
    networks:
      - app-network
```

Some of the settings we defined for the `nodejs` service remain the same, but we've also made the following changes:

- `image`: This tells Compose to pull the latest Alpine-based Nginx image from Docker Hub. For more information about `alpine` images, please see Step 3 of How To Build a Node.js

- `web-root:/var/www/html` : This will add our site's static assets, copied to a volume called `web-root` , to the the `/var/www/html` directory on the container.

- `certbot-var:/var/lib/letsencrypt` : This mounts Let's Encrypt's default working directory to the appropriate directory on the container.

Next, add the configuration options for the `certbot` container. Be sure to replace the domain and email information with your own domain name and contact email:

~/node_project/docker-compose.yml

```
...
  certbot:
    image: certbot/certbot
    container_name: certbot
    volumes:
      - certbot-etc:/etc/letsencrypt
      - certbot-var:/var/lib/letsencrypt
      - web-root:/var/www/html
    depends_on:
      - webserver
    command: certonly --webroot --webroot-path=/var/www/html --email sammy@example.com -
example.com
```

This definition tells Compose to pull the certbot/certbot image from Docker Hub. It also uses named volumes to share resources with the Nginx container, including the domain certificates and key in `certbot-etc` , the Let's Encrypt working directory in `certbot-var` , and the application code in `web-root` .

Again, we've used `depends_on` to specify that the `certbot` container should be started once the `webserver` service is running.

- `--webroot-path` : This specifies the path of the webroot directory.
- `--email` : Your preferred email for registration and recovery.

configuration options and avoid possible domain request limits. For more information about these limits, please see Let's Encrypt's rate limits documentation.

- `-d` : This allows you to specify domain names you would like to apply to your request. In this case, we've included `example.com` and `www.example.com` . Be sure to replace these with your own domain preferences.

As a final step, add the volume and network definitions. Be sure to replace the username here with your own non-root user:

~/node_project/docker-compose.yml

```
...
volumes:
  certbot-etc:
  certbot-var:
  web-root:
    driver: local
    driver_opts:
      type: none
      device: /home/sammy/node_project/views/
      o: bind

networks:
  app-network:
    driver: bridge
```

Our named volumes include our Certbot certificate and working directory volumes, and the volume for our site's static assets, `web-root` . In most cases, the default driver for

The `docker-compose.yml` file will look like this when finished:


```
      dockerfile: Dockerfile
    image: nodejs
    container_name: nodejs
    restart: unless-stopped
    networks:
      - app-network

  webserver:
    image: nginx:mainline-alpine
    container_name: webserver
    restart: unless-stopped
    ports:
      - "80:80"
    volumes:
      - web-root:/var/www/html
      - ./nginx-conf:/etc/nginx/conf.d
      - certbot-etc:/etc/letsencrypt
      - certbot-var:/var/lib/letsencrypt
    depends_on:
      - nodejs
    networks:
      - app-network

  certbot:
    image: certbot/certbot
    container_name: certbot
    volumes:
      - certbot-etc:/etc/letsencrypt
      - certbot-var:/var/lib/letsencrypt
      - web-root:/var/www/html
    depends_on:
      - webserver
    command: certonly --webroot --webroot-path=/var/www/html --email sammy@example.com -
```

```
networks:
  app-network:
```


Step 4 — Obtaining SSL Certificates and Credentials

We can start our containers with `docker-compose up`, which will create and run our containers and services in the order we have specified. If our domain requests are successful, we will see the correct exit status in our output and the right certificates mounted in the `/etc/letsencrypt/live` folder on the `webserver` container.

Create the services with `docker-compose up` and the `-d` flag, which will run the `nodejs` and `webserver` containers in the background:

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

You will see output confirming that your services have been created:

```
Output
Creating nodejs ... done
Creating webserver ... done
Creating certbot   ... done
```

Using `docker-compose ps`, check the status of your services:

```
$ docker-compose ps
```

If everything was successful, your `nodejs` and `webserver` services should be `Up` and the

```
nodejs      node app.js                  Up      8080/tcp
webserver   nginx -g daemon off;         Up      0.0.0.0:80->80/tcp
```


You can now check that your credentials have been mounted to the `webserver` container with `docker-compose exec`:

```
$ docker-compose exec webserver ls -la /etc/letsencrypt/live
```

If your request was successful, you will see output like this:

```
Output
total 16
drwx------ 3 root root 4096 Dec 23 16:48 .
drwxr-xr-x 9 root root 4096 Dec 23 16:48 ..
-rw-r--r-- 1 root root  740 Dec 23 16:48 README
drwxr-xr-x 2 root root 4096 Dec 23 16:48 example.com
```

Now that you know your request will be successful, you can edit the `certbot` service definition to remove the `--staging` flag.

Open `docker-compose.yml`:

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

Find the section of the file with the `certbot` service definition, and replace the `--staging` flag in the `command` option with the `--force-renewal` flag, which will tell

```
    image: certbot/certbot
    container_name: certbot
    volumes:
```


You can now run `docker-compose up` to recreate the `certbot` container and its relevant volumes. We will also include the `--no-deps` option to tell Compose that it can skip starting the `webserver` service, since it is already running:

```
$ docker-compose up --force-recreate --no-deps certbot
```

You will see output indicating that your certificate request was successful:

```
Output
certbot      | IMPORTANT NOTES:
certbot      |  - Congratulations! Your certificate and chain have been saved at:
certbot      |    /etc/letsencrypt/live/example.com/fullchain.pem
certbot      |    Your key file has been saved at:
certbot      |    /etc/letsencrypt/live/example.com/privkey.pem
certbot      |    Your cert will expire on 2019-03-26. To obtain a new or tweaked
certbot      |    version of this certificate in the future, simply run certbot
certbot      |    again. To non-interactively renew *all* of your certificates, run
certbot      |    "certbot renew"
certbot      |  - Your account credentials have been saved in your Certbot
certbot      |    configuration directory at /etc/letsencrypt. You should make a
certbot      |    secure backup of this folder now. This configuration directory will
certbot      |    also contain certificates and private keys obtained by Certbot so
certbot      |    making regular backups of this folder is ideal.
certbot      |  - If you like Certbot, please consider supporting our work by:
certbot      |
```

Step 5 — Modifying the Web Server Configuration and Service Definition

Enabling SSL in our Nginx configuration will involve adding an HTTP redirect to HTTPS


```
$ docker-compose stop webserver
```

Next, create a directory in your current project directory for your Diffie-Hellman key:

```
$ mkdir dhparam
```

Generate your key with the `openssl` command:

```
$ sudo openssl dhparam -out /home/sammy/node_project/dhparam/dhparam-2048.pem 2048
```

It will take a few moments to generate the key.

To add the relevant Diffie-Hellman and SSL information to your Nginx configuration, first remove the Nginx configuration file you created earlier:

```
$ rm nginx-conf/nginx.conf
```

Open another version of the file:

```
$ nano nginx-conf/nginx.conf
```

Add the following code to the file to redirect HTTP to HTTPS and to add SSL credentials,

```
        server_name example.com www.example.com;

        location ~ /.well-known/acme-challenge {
```



```
server {
        listen 443 ssl http2;
        listen [::]:443 ssl http2;
        server_name example.com www.example.com;

        server_tokens off;

        ssl_certificate /etc/letsencrypt/live/example.com/fullchain.pem;
        ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;

        ssl_buffer_size 8k;

        ssl_dhparam /etc/ssl/certs/dhparam-2048.pem;

        ssl_protocols TLSv1.2 TLSv1.1 TLSv1;
        ssl_prefer_server_ciphers on;

        ssl_ciphers ECDH+AESGCM:ECDH+AES256:ECDH+AES128:DH+3DES:!ADH:!AECDH:!MD5;

        ssl_ecdh_curve secp384r1;
        ssl_session_tickets off;

        ssl_stapling on;
        ssl_stapling_verify on;
        resolver 8.8.8.8;

        location / {
                try_files $uri @nodejs;
        }

        location @nodejs {
                proxy_pass http://nodejs:8080;
                add_header X-Frame-Options "SAMEORIGIN" always;
```

```
        index index.html index.htm index.nginx-debian.html;
}
```


iterates on HTTP protocols and the benefits it can have for website performance, please see the introduction to How To Set Up Nginx with HTTP/2 Support on Ubuntu 18.04. This block also includes a series of options to ensure that you are using the most up-to-date SSL protocols and ciphers and that OSCP stapling is turned on. OSCP stapling allows you to offer a time-stamped response from your certificate authority during the initial TLS handshake, which can speed up the authentication process.

The block also specifies your SSL and Diffie-Hellman credentials and key locations.

Finally, we've moved the proxy pass information to this block, including a location block with a `try_files` directive, pointing requests to our aliased Node.js application container, and a location block for that alias, which includes security headers that will enable us to get **A** ratings on things like the SSL Labs and Security Headers server test sites. These headers include `X-Frame-Options`, `X-Content-Type-Options`, `Referrer Policy`, `Content-Security-Policy`, and `X-XSS-Protection`. The HTTP `Strict Transport Security` (HSTS) header is commented out — enable this only if you understand the implications and have assessed its "preload" functionality.

Once you have finished editing, save and close the file.

Before recreating the `webserver` service, you will need to add a few things to the service definition in your `docker-compose.yml` file, including relevant port information for HTTPS and a Diffie-Hellman volume definition.

Open the file:

~/node_project/docker-compose.yml


```
      - web-root:/var/www/html
      - ./nginx-conf:/etc/nginx/conf.d
      - certbot-etc:/etc/letsencrypt
      - certbot-var:/var/lib/letsencrypt
      - dhparam:/etc/ssl/certs
    depends_on:
      - nodejs
    networks:
      - app-network
```

Next, add the `dhparam` volume to your `volumes` definitions:

~/node_project/docker-compose.yml

```
...
volumes:
  ...
  dhparam:
    driver: local
    driver_opts:
      type: none
      device: /home/sammy/node_project/dhparam/
      o: bind
```

Similarly to the `web-root` volume, the `dhparam` volume will mount the Diffie-Hellman key stored on the host to the `webserver` container.

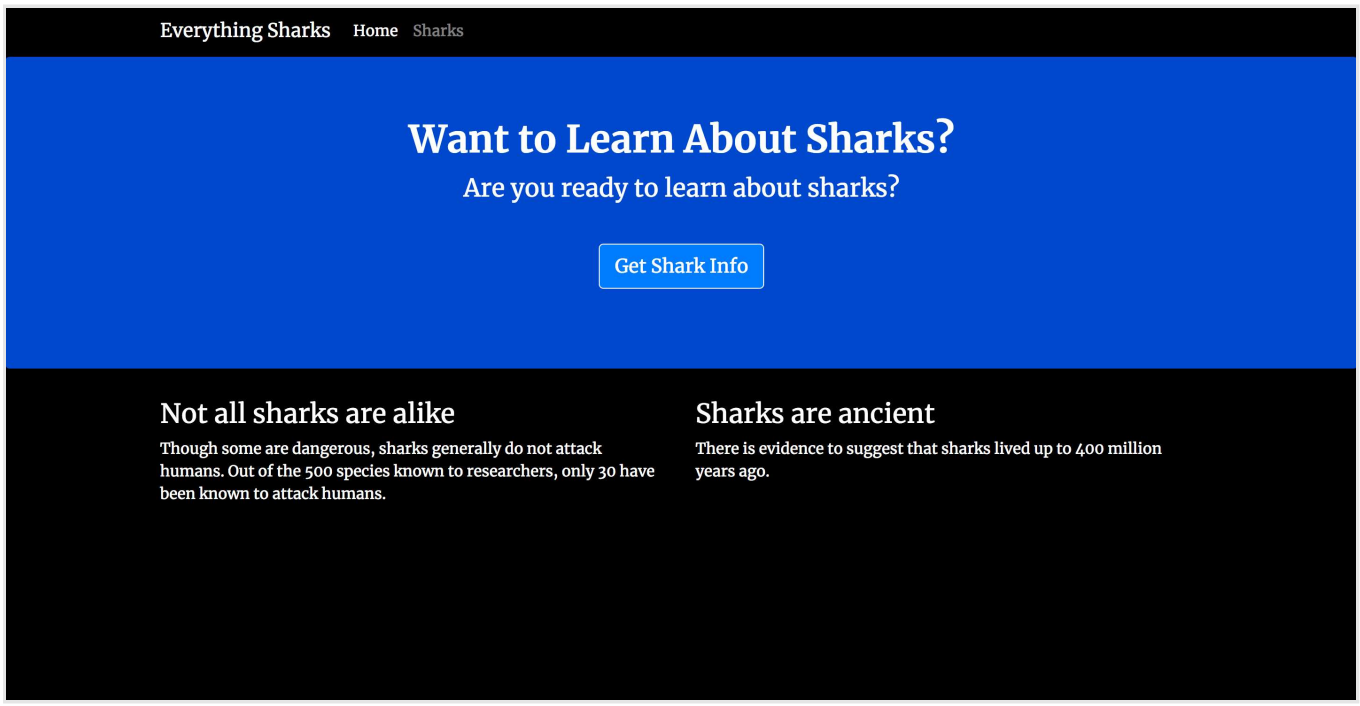
Save and close the file when you are finished editing.

```
$ docker-compose ps
```



```
nodejs      node app.js                    Up      8080/tcp
webserver   nginx -g daemon off;           Up      0.0.0.0:443->443/tcp, 0.0.0.0:80->80/
```

Finally, you can visit your domain to ensure that everything is working as expected. Navigate your browser to `https://example.com`, making sure to substitute `example.com` with your own domain name. You will see the following landing page:



You should also see the lock icon in your browser's security indicator. If you would like, you can navigate to the SSL Labs Server Test landing page or the Security Headers server test landing page. The configuration options we've included should earn your site

with the `cron` scheduling utility. In this case, we will schedule a `cron` job using a script that will renew our certificates and reload our Nginx configuration.

server configuration:

~/node_project/ssl_renew.sh

```
#!/bin/bash

COMPOSE="/usr/local/bin/docker-compose --no-ansi"
DOCKER="/usr/bin/docker"

cd /home/sammy/node_project/
$COMPOSE run certbot renew --dry-run && $COMPOSE kill -s SIGHUP webserver
$DOCKER system prune -af
```

This script first assigns the `docker-compose` binary to a variable called `COMPOSE`, and specifies the `--no-ansi` option, which will run `docker-compose` commands without ANSI control characters. It then does the same with the `docker` binary. Finally, it changes to the `~/node_project` directory and runs the following `docker-compose` commands:

- `docker-compose run`: This will start a `certbot` container and override the `command` provided in our `certbot` service definition. Instead of using the `certonly` subcommand, we're using the `renew` subcommand here, which will renew certificates that are close to expiring. We've included the `--dry-run` option here to test our script.
- `docker-compose kill`: This will send a `SIGHUP` signal to the `webserver` container to reload the Nginx configuration. For more information on using this process to reload your Nginx configuration, please see this Docker blog post on deploying the official Nginx image with Docker.

Next, open your **root** `crontab` file to run the renewal script at a specified interval:


```
no crontab for root - using an empty one
Select an editor.  To change later, run 'select-editor'.
  1. /bin/ed
  2. /bin/nano        <---- easiest
  3. /usr/bin/vim.basic
  4. /usr/bin/vim.tiny
Choose 1-4 [2]:
...
```

At the bottom of the file, add the following line:

crontab

```
...
*/5 * * * * /home/sammy/node_project/ssl_renew.sh >> /var/log/cron.log 2>&1
```

This will set the job interval to every five minutes, so you can test whether or not your renewal request has worked as intended. We have also created a log file, `cron.log`, to record relevant output from the job.

After five minutes, check `cron.log` to see whether or not the renewal request has succeeded:

```
$ tail -f /var/log/cron.log
```

You should see output confirming a successful renewal:

```
  /etc/letsencrypt/live/example.com/fullchain.pem (success)
** DRY RUN: simulating 'certbot renew' close to cert expiry
**          (The test certificates above have not been saved.)
```


crontab

```
...
0 12 * * * /home/sammy/node_project/ssl_renew.sh >> /var/log/cron.log 2>&1
```

You will also want to remove the `--dry-run` option from your `ssl_renew.sh` script:

~/node_project/ssl_renew.sh

```
#!/bin/bash

COMPOSE="/usr/local/bin/docker-compose --no-ansi"
DOCKER="/usr/bin/docker"

cd /home/sammy/node_project/
$COMPOSE run certbot renew && $COMPOSE kill -s SIGHUP webserver
$DOCKER system prune -af
```

Your `cron` job will ensure that your Let's Encrypt certificates don't lapse by renewing them when they are eligible. You can also set up log rotation with the Logrotate utility to rotate and compress your log files.

Conclusion

You have used containers to set up and run a Node application with an Nginx reverse proxy. You have also secured SSL certificates for your application's domain and set up a `cron` job to renew these certificates when necessary.

- How To Configure a Continuous Integration Testing Environment with Docker and Docker Compose on Ubuntu 16.04.

About the authors

Kathleen Juell

Developer
@digitalocean/community

Tutorial Series

From Containers to Kubernetes with Node.js

In this series, you will build and containerize a Node.js application with a MongoDB database. The series is designed to introduce you to the fundamentals of migrating an application to Kubernetes, including modernizing your app using the 12FA methodology, containerizing it, and

Still looking for an answer?

RELATED

How To Work With Zip Files in Node.js

Tutorial

How To Configure Suricata as an Intrusion Prevention System (IPS) on Rocky Linux 8

Tutorial

Comments

57 Comments

Leave a comment...

Sign In to Comment

jasper February 3, 2019

2 By the way, if you have environmental variables in your `docker-compose.yml` the whole setup will not work as cron cannot read these vars properly.

Reply Report

christianstrang April 10, 2019

2 Don't be like me, don't forget to set "–force-renewal" in docker-compose.yml (I lost way to much time than I care to admit).

Reply Report

> RonAlmog August 18, 2020
>
> 0 Where? i don't see this in the article. They do the renewal of ssl with a cron job. please explain.
>
> Reply Report
>
> > tkpremier October 8, 2020
> >
> > 0 I believe OP is talking about Step 5, when you add the 'certbot' block to docker-compose.yml
> >
> > You have to scroll horizontally to realize that there's a "force-renewal" param added :)
> >
> > Reply Report

philcockfield April 22, 2019

5 References to `create xxx in current directory` is ambiguous.

2 Just a quick question, If I would like to make a change in app.js file because I am using nodejs as a server proxy RESTful APIs what can I do with volume to update everytime I

1 **rabierambroise** June 16, 2019

Very helpful tutorial. I am actually trying to add a subdomain in a different server block on nginx. But I get privacy error on chrome. Is there something else todo apart from adding `-d subdomain.mywebsite.com` ?

Got this working:

```
server {
        listen 80;
        listen [::]:80;
# sub domain added here
        server_name ambroise-rabier.fr www.ambroise-rabier.fr analytics.ambroise-ra

        location ~ /.well-known/acme-challenge {
          allow all;
          root /var/www/html;
        }

        location / {
                rewrite ^ https://$host$request_uri? permanent;
        }
}
```

```
server {
    listen 443 ssl http2;
```

```
        ssl_certificate /etc/letsencrypt/live/ambroise-rabier.fr/fullchain.pem;
        ssl_certificate_key /etc/letsencrypt/live/ambroise-rabier.fr/privkey.pem;
```



```
        ssl_ciphers ECDH+AESGCM:ECDH+AES256:ECDH+AES128:DH+3DES:!ADH:!AECDH:!MD5;

        ssl_ecdh_curve secp384r1;
        ssl_session_tickets off;

        ssl_stapling on;
        ssl_stapling_verify on;
        resolver 8.8.8.8;

        root /var/www/html/matomo;

        index index.html;

        location / {
            try_files $uri $uri/ =404;
        }

    }
```

Now I get:

```
2019/06/16 09:23:57 [warn] 67#67: conflicting server name "ambroise-rabier.fr" on 0
nginx: [warn] conflicting server name "ambroise-rabier.fr" on 0.0.0.0:443, ignored
2019/06/16 09:23:57 [warn] 67#67: conflicting server name "ambroise-rabier.fr" on [
nginx: [warn] conflicting server name "ambroise-rabier.fr" on [::]:443, ignored
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

1 Great tutorial! Had some issues that were mostly self-created, non-root user and references to folders that didn't exist, but otherwise this tutorial allowed me to create an Express.js


```
location / {
    proxy_pass http://nodejs:8080;
}
```

Reply Report

ItaiB August 13, 2019

2 it's the container_name: nodejs, that she put in the docker-compose.yml

Reply Report

adamfairweather32 November 3, 2019

1 Hi, can you please upload the completed project to github? I can't seem to get mine to work once I try and enable https

Reply Report

ss729 April 12, 2020

0 I had this issue and was able to resolve it by allowing HTTPS on port 443 in my firewall settings - hope this helps!

Reply Report

flowster November 7, 2019

1 It's too bad this doesn't tie in with your previous tutorial in the series (https://www.digitalocean.com/community/tutorials/how-to-scale-a-node-js-application-with-mongodb-on-kubernetes-using-helm), and explain how to secure the application with Kubernetes ... any articles on that?

Reply Report

stewshadow December 31, 2019

0 I don't know, and I wish the author could relate to this.
So what i plan to do is: after completing step 5, (real certificate works), i'm going to comment the whole certbot part of the docker-compose.
i don't need it any more, right? so i'll keep it in comments.

Reply Report

syncnsecure April 14, 2020

0 As soon as i port from http to https everything stops working. I have tried it several times with many tutorials however no luck.

This site can't be reached www.dashboard.sns-hub.com's server IP address could not be found.
Try running Windows Network Diagnostics.
DNS*PROBE*FINISHED_NXDOMAIN

certbot | Saving debug log to /var/log/letsencrypt/letsencrypt.log
certbot | Plugins selected: Authenticator webroot, Installer None
certbot | Renewing an existing certificate
certbot | Performing the following challenges:
certbot | http-01 challenge for www.dashboard.sns-hub.com
certbot | Using the webroot path /var/www/html for all unmatched domains.
certbot | Waiting for verification...
certbot | Challenge failed for domain www.dashboard.sns-hub.com
certbot | http-01 challenge for www.dashboard.sns-hub.com
certbot | Cleaning up challenges
certbot | Some challenges have failed.
certbot | IMPORTANT NOTES:

certbot | domain
certbot exited with code 1

serranomorantepatricio May 16, 2020

0

It works! Oh my god. Thank you so much.
I followed the tutorial but instead of Node I was using Django the whole time.

Reply Report

mikeng May 28, 2020

0

Thanks, nice tutorial. Follow up question:

so every time the live renewal cron runs (every 12 hours here in this example) it will kill the nginx service whether certbot renews the certs or not ?

This is certainly unwanted. We do not want to interrupt the accessibility of our websites unless we have to and the certs need renewal...

Any way to only kill nginx service when we actually renew the certs ?

Reply Report

> katjuell May 28, 2020
>
> 0
>
> Hey @mikeng – using `docker-compose kill` with the the `SIGHUP` signal will reload your nginx config. You can check out this blog post for more details. Hope that helps.
>
> Reply Report
>
> > mikeng May 29, 2020
> >
> > 0
> >
> > Kathleen,
> >
> > thanks for elaborating on that. Great tutorial, very detailed.
> >
> > - M

eof March 6, 2021

0 in my solution I've changed SIGHUP into:

0 Hi folks

This is a very basic question but here it goes: do I need to run the tutorial on the webserver? I have this domain *heykidhealth.com* registered in AWS Route 53 and the website is a static one under S3, basically a draft. If I have to execute the tutorial on the webserver, it would not be possible due to it is a bucket and I don't even know if this is allowed by AWS.
Anyway, as I assumed that it could be run from any server, I created a very basic one from which I executed the tutorial but everything went wrong.
This is the log I've got from certbot:

```
docker-compose logs certbot
Attaching to certbot
certbot      | Saving debug log to /var/log/letsencrypt/letsencrypt.log
certbot      | Plugins selected: Authenticator webroot, Installer None
certbot      | Obtaining a new certificate
certbot      | Performing the following challenges:
certbot      | http-01 challenge for heykidhealth.com
certbot      | http-01 challenge for www.heykidhealth.com
certbot      | Using the webroot path /var/www/html for all unmatched domains.
certbot      | Waiting for verification...
certbot      | Challenge failed for domain heykidhealth.com
certbot      | Challenge failed for domain www.heykidhealth.com
certbot      | http-01 challenge for heykidhealth.com
certbot      | http-01 challenge for www.heykidhealth.com
certbot      | Cleaning up challenges
certbot      | Some challenges have failed.
certbot      | IMPORTANT NOTES:
certbot      |  - The following errors were reported by the server:
```

```
certbot |   Domain: www.heykidhealth.com
certbot |   Type:   unauthorized
```



```
certbot |   contain(s) the right IP address.
```

My goal is to create a wildcard Let's Encrypt certificate and register it on AWS so all my servers could get certified at once, webserver included, mainly API & database servers. For now, I would be very glad if I manage to create one certificate.

Thanks in advance.
Renato

Reply Report

SandyCyanSnorkler June 13, 2020

0

Same as me, I'm facing this issue! Somebody please help us.

Reply Report

renatospaka June 17, 2020

0

Hey @SandyCyanSnorkler, how are you doing?

I changed my approach and headed to acme.sh in order to generate my wildcard certificate and it worked just fine. It uses Let's Encrypt, it is fast and very straight-forward, took less than 1 minute to finish the process to generate the certificate, and it scheduled a revalidation process with cron on my behalf every 60 days.

However, I didn't use Docker despite it is Docker friendly. If you learn how to do that with Docker, please let me know.

Renato

1 hi, thanks for the solid guide. My problem is ssl_renewal.sh script you suggested, namely ending line: docker system prune -af

cyrilcabo August 15, 2020

1 My app stops working after step 5. Basically the nginx configuration for HTTPS keeps my application from working. Can anyone please help me?

Here's my nginx configuration: https://github.com/cyrilcabo/alphadevelopment/blob/master/nginx-conf/nginx.conf

And here's my docker-compose.yml: https://github.com/cyrilcabo/alphadevelopment/blob/master/docker-compose.yml

Reply Report

kwabenaadudarkwa August 15, 2020

0 I am getting this error in websever logs:

[error] 19#19: *1 connect() failed (111: Connection refused) while connecting to upstream, client: XXX.XXX.XX.XXX, server: [domain name], request: "GET /favicon.ico HTTP/2.0", upstream: "http://XXX.XXX.XX.X:5000/favicon.ico", host: [domain name], referrer: "https://[domain name]/"

Any help?

Reply Report

> bobbyiliev August 17, 2020
>
> 0 Hello,
>
> You need to check if your backend service is running on port `5000`. To do that run this command here:

Regards,
Bobby

bobbyiliev August 17, 2020

0

Hi @kwabenaadudarkwa,

No problem, happy to hear that you've got it all working!

Regards,
Bobby

Reply Report

Load More Comments

Release Notes