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

7

Posted January 4, 2019 4.3k DOCKER NODE.JS LET'S ENCRYPT SECURITY UBUNTU 18.04

By: Kathleen Juell

Introduction

There are multiple ways to enhance the flexibility and security of your Node.js application. Using a reverse proxy like Nginx offers you the ability to load balance requests, cache static content, and implement *Transport Layer Security* (TLS). Enabling encrypted HTTPS on your server ensures that communication to and from your application remains secure.

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 obtaining certificates from Let's Encrypt for an application running on a server, you would install the required software directly on your host. Containers allow you to take a different approach. Using Docker Compose, you can create containers for your application, your web server, and the Certbot client that will enable you to obtain your certificates. By following these steps, you can take advantage of the modularity and portability of a containerized workflow.

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.

Step 1 — Cloning and Testing the Node Application

As a first step, we will clone the repository with the Node application code, which includes the Dockerfile that we will use to build our application image with Compose. We can first test the application by building and running it with the `docker run` command, without a reverse proxy or SSL.

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 setup described in How To Build a Node.js Application with Docker.

Clone the repository into a directory called `node_project`:

```
$ git clone https://github.com/do-community/nodejs-image-demo.git node_project
```

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 ./

RUN npm install
```

```
COPY . .

COPY --chown=node:node . .

USER node

EXPOSE 8080

CMD [ "node", "app.js" ]
```

These instructions build a Node image by copying the project code from the current directory to the container and installing dependencies with `npm install`. They also take advantage of Docker's caching and image layering by separating the copy of `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 container will be run as the non-root **node** user with the appropriate permissions set on the application code and `node_modules` directories.

For more information about this Dockerfile and Node image best practices, please see the complete discussion in Step 3 of How To Build a Node.js Application with Docker.

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.
- `-d`: This runs the container in the background.
- `--name`: This allows us to give the container a memorable name.

Run the following command to build the container:

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

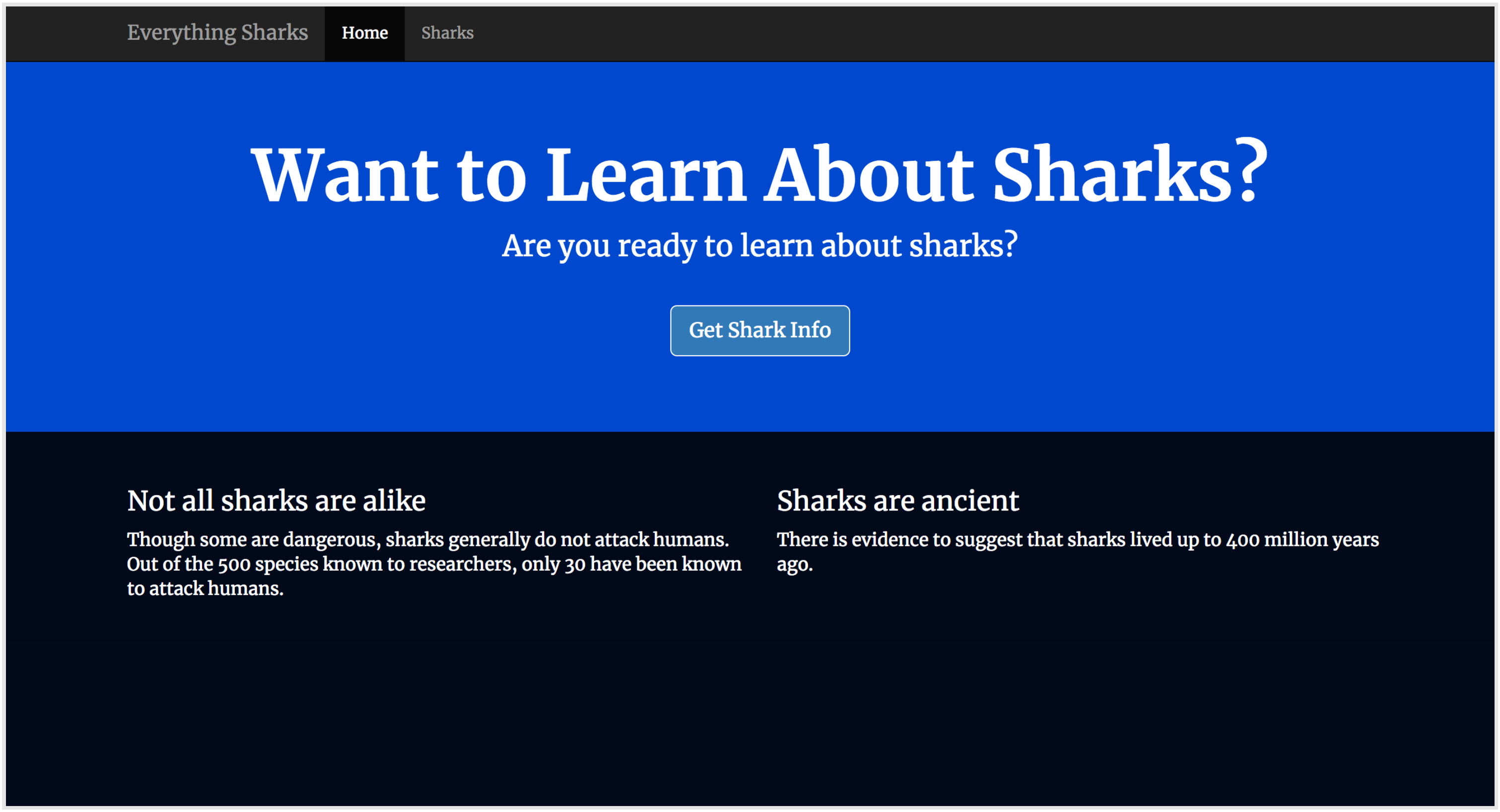
Inspect your running containers with `docker ps`:

```
$ docker ps
```

You will see output confirming that your application container is running:

```
Output
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              P
4133b72391da        node-demo           "node app.js"       17 seconds ago      Up 16 seconds       0
```

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:



Want to Learn About Sharks?

Are you ready to learn about sharks?

Not all sharks are alike

Though some are dangerous, sharks generally do not attack humans. Out of the 500 species known to researchers, only 30 have been known to attack humans.

Sharks are ancient

There is evidence to suggest that sharks lived up to 400 million years ago.

Now that you have tested the application, you can stop the container and remove the images. Use `docker ps` again to get your `CONTAINER ID`:

```
$ docker ps
```

```
Output
```

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              P
4133b72391da        node-demo           "node app.js"       17 seconds ago      Up 16 seconds       0
```

Stop the container with `docker stop`. Be sure to replace the `CONTAINER ID` listed here with your own application `CONTAINER ID`:

```
$ docker stop 4133b72391da
```

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:

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

Add the following server block to proxy user requests to your Node application container and to direct Certbot's requests to the `.well-known` directory. Be sure to replace `example.com` with your own domain name:

~/node_project/nginx-conf/nginx.conf

```
server {
        listen 80;
        listen [::]:80;
        root /var/www/html;
        index index.html index.htm index.nginx-debian.html;

        server_name example.com www.example.com;

        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 web server. It will specify details like named volumes, which will be critical to sharing SSL credentials between containers, as well as network and port information. It will also allow us to specify specific commands to run when our containers are created. This file is the central resource that will define how our services will work together.

Open the file in your current directory:

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

First, define the application service:

~/node_project/docker-compose.yml

```
version: '3'

services:
  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.

To enable communication between the application and web server containers, we will also add a bridge network called `app-network` below the restart definition:

~/node_project/docker-compose.yml

```
services:
  nodejs:
...
    networks:
      - app-network
```

A user-defined bridge network like this enables communication between containers on the same Docker daemon host. This streamlines traffic and communication within your application, since it opens all ports between containers on the same bridge network, 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 Application with Docker.
- `ports`: This exposes port `80` to enable the configuration options we've defined in our Nginx configuration.

We have also specified the following named volumes and bind mounts:

- `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.
- `./nginx-conf:/etc/nginx/conf.d`: This will bind mount the Nginx configuration directory on the host to the relevant directory on the container, ensuring that any changes we make to files on the host will be reflected in the container.
- `certbot-etc:/etc/letsencrypt`: This will mount the relevant Let's Encrypt certificates and keys for our domain to the appropriate 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 --agree-tos --
```

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.

We've also included a `command` option that specifies the command to run when the container is started. It includes the `certonly` subcommand with the following options:

- `--webroot`: This tells Certbot to use the webroot plugin to place files in the webroot folder for authentication.
- `--webroot-path`: This specifies the path of the webroot directory.
- `--email`: Your preferred email for registration and recovery.
- `--agree-tos`: This specifies that you agree to ACME's Subscriber Agreement.
- `--no-eff-email`: This tells Certbot that you do not wish to share your email with the Electronic Frontier Foundation (EFF). Feel free to omit this if you would prefer.
- `--staging`: This tells Certbot that you would like to use Let's Encrypt's staging environment to obtain test certificates. Using this option allows you to test your 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 Docker volumes is the `local` driver, which on Linux accepts options similar to the `mount` command. Thanks to this, we are able to specify a list of driver options with `driver_opts` that mount the `views` directory on the host, which contains our application's static assets, to the volume at runtime. The directory contents can then be shared between containers. For more information about the contents of the `views` directory, please see Step 2 of How To Build a Node.js Application with Docker.

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

~/node_project/docker-compose.yml

```
version: '3'

services:
  nodejs:
    build:
      context: .
      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 --agree-tos --

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
```

With the service definitions in place, you are ready to start the containers and test your certificate requests.

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 `certbot` container will have exited with a `0` status message:

```
Output
  Name                 Command               State          Ports
----------------------------------------------------------------------
certbot     certbot certonly --webroot ...   Exit 0
nodejs      node app.js                      Up       8080/tcp
webserver   nginx -g daemon off;             Up       0.0.0.0:80->80/tcp
```

If you see anything other than `Up` in the `State` column for the `nodejs` and `webserver` services, or an exit status other than `0` for the `certbot` container, be sure to check the service logs with the `docker-compose logs` command:

```
$ docker-compose logs service_name
```

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 Certbot that you want to request a new certificate with the same domains as an existing certificate. The `certbot` service definition should now look like this:

~/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 --agree-tos -
...
```

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      |
certbot      |    Donating to ISRG / Let's Encrypt:   https://letsencrypt.org/donate
certbot      |    Donating to EFF:                    https://eff.org/donate-le
certbot      |
certbot exited with code 0
```

With your certificates in place, you can move on to modifying your Nginx configuration to include SSL.

Step 5 — Modifying the Web Server Configuration and Service Definition

Enabling SSL in our Nginx configuration will involve adding an HTTP redirect to HTTPS and specifying our SSL certificate and key locations. It will also involve specifying our Diffie-Hellman group, which we will use for Perfect Forward Secrecy.

Since you are going to recreate the `webserver` service to include these additions, you can stop it now:

```
$ 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, protocols, and security headers. Remember to replace `example.com` with your own domain:

~/node_project/nginx-conf/nginx.conf

```
server {
        listen 80;
        listen [::]:80;
        server_name example.com www.example.com;

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

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

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;
                add_header X-XSS-Protection "1; mode=block" always;
                add_header X-Content-Type-Options "nosniff" always;
                add_header Referrer-Policy "no-referrer-when-downgrade" always;
                add_header Content-Security-Policy "default-src * data: 'unsafe-eval' 'unsafe-inline'
                # add_header Strict-Transport-Security "max-age=31536000; includeSubDomains; preload'
                # enable strict transport security only if you understand the implications
        }

        root /var/www/html;
        index index.html index.htm index.nginx-debian.html;
}
```

The HTTP server block specifies the webroot for Certbot renewal requests to the `.well-known/acme-challenge` directory. It also includes a rewrite directive that directs HTTP requests to the root directory to HTTPS.

The HTTPS server block enables `ssl` and `http2`. To read more about how HTTP/2 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:

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

In the `webserver` service definition, add the following port mapping and the `dhparam` named volume:

~/node_project/docker-compose.yml

```
...
 webserver:
    image: nginx:latest
    container_name: webserver
    restart: unless-stopped
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - 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.

Recreate the `webserver` service:

```
$ docker-compose up -d --force-recreate --no-deps webserver
```

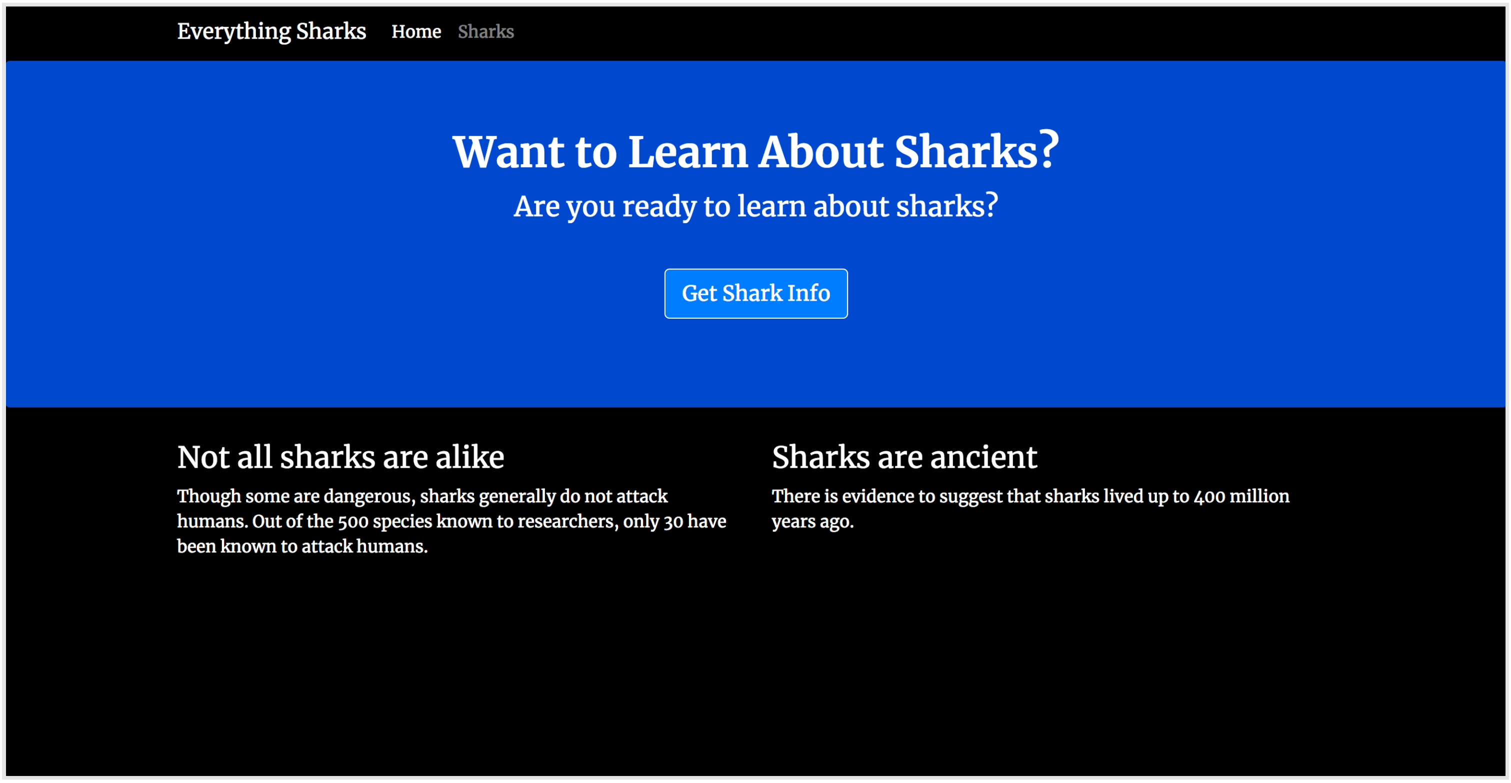
Check your services with `docker-compose ps`:

```
$ docker-compose ps
```

You should see output indicating that your `nodejs` and `webserver` services are running:

```
Output
  Name                   Command                     State                          Ports
-----------------------------------------------------------------------------------------------
certbot     certbot certonly --webroot       Exit 0
certbot     certbot certonly --webroot ...   Exit 0
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/tcp
```

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:



Want to Learn About Sharks?

Are you ready to learn about sharks?

Not all sharks are alike

Though some are dangerous, sharks generally do not attack humans. Out of the 500 species known to researchers, only 30 have been known to attack humans.

Sharks are ancient

There is evidence to suggest that sharks lived up to 400 million years ago.

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 an **A** rating on both.

Step 6 — Renewing Certificates

Let's Encrypt certificates are valid for 90 days, so you will want to set up an automated renewal process to ensure that they do not lapse. One way to do this is to create a job 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.

Open a script called `ssl_renew.sh` in your project directory:

```
$ nano ssl_renew.sh
```

Add the following code to the script to renew your certificates and reload your web server configuration:

~/node_project/ssl_renew.sh

```
#!/bin/bash
```

```
/usr/local/bin/docker-compose -f /home/sammy/node_project/docker-compose.yml run certbot renew --dry-
&& /usr/local/bin/docker-compose -f /home/sammy/node_project/docker-compose.yml kill -s SIGHUP webser
```

In addition to specifying the location of our `docker-compose` binary, we also specify the location of our `docker-compose.yml` file in order to run `docker-compose` commands. In this case, we are using `docker-compose run` to start a `certbot` container and to override the `command` provided in our service definition with another: the `renew` subcommand, which will renew certificates that are close to expiring. We've included the `--dry-run` option here to test our script.

The script then uses `docker-compose kill` to 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.

Close the file when you are finished editing. Make it executable:

```
$ chmod +x ssl_renew.sh
```

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

```
$ sudo crontab -e
```

If this is your first time editing this file, you will be asked to choose an editor:

crontab

```
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:

```
Output
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
** DRY RUN: simulating 'certbot renew' close to cert expiry
**          (The test certificates below have not been saved.)


Congratulations, all renewals succeeded. The following certs have been renewed:
  /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.)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Killing webserver ... done
```

You can now modify the `crontab` file to set a daily interval. To run the script every day at noon, for example, you would modify the last line of the file to look like this:

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

/usr/local/bin/docker-compose -f /home/sammy/node_project/docker-compose.yml run certbot renew \
&& /usr/local/bin/docker-compose -f /home/sammy/node_project/docker-compose.yml kill -s SIGHUP webser
```

Your `cron` job will ensure that your Let's Encrypt certificates don't lapse by renewing them when they are eligible.

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.

If you are interested in learning more about Let's Encrypt plugins, please see our articles on using the Nginx plugin or the standalone plugin.

You can also learn more about Docker Compose by looking at the following resources:

- How To Install Docker Compose on Ubuntu 18.04.
- How To Configure a Continuous Integration Testing Environment with Docker and Docker Compose on Ubuntu 16.04.
- How To Set Up Laravel, Nginx, and MySQL with Docker Compose.

The Compose documentation is also a great resource for learning more about multi-container applications.

By: Kathleen Juell

Related Tutorials

How To Sync and Share Your Files with Seafile on Debian 9

How To Install YunoHost on Debian 9

How To Ensure Code Quality with SonarQube on Ubuntu 18.04

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

3 Comments

Leave a comment...

Log In to Comment

jasper *January 8, 2019*

0 Thanks for this wonderful tutorial. Learned a lot from it working on my Laradock setup at https://github.com/Larastudio/lsdock . This post helped me get moving with Certbot. Nginx tweaks and deployment structure are up next.

whytellingyou *January 14, 2019*

0 Roboform keygen is an excellent password managing tool. Once it saves the password there is no need to write password again and again. It remembers the passwords and keeps secure. You simply have to get registration to get an ID. You log on and enter your important passwords and keep them safe in the system. Along with the security of your important passwords it also keeps your internet security. Your all personal information is saved and there are fewer chances of leakage of your personal data. You can save different usernames and passwords with confidence.

teleworm1337 *January 18, 2019*

0 Implementing a reverse proxy with TLS/SSL on containers involves a different set of procedures from working directly on a host operating system. https://downloadluckypatcher.co