

How To Build a Node.js Application with Docker

Posted November 29, 2018  21.9k

DOCKER

NODE.JS

APPLICATIONS

UBUNTU 18.04



By: Kathleen Juell

Introduction

The Docker platform allows developers to package and run applications as *containers*. A container is an isolated process that runs on a shared operating system, offering a lighter weight alternative to virtual machines. Though containers are not new, they offer benefits — including process isolation and environment standardization — that are growing in importance as more developers use distributed application architectures.

When building and scaling an application with Docker, the starting point is typically creating an image for your application, which you can then run in a container. The image includes your application code, libraries, configuration files, environment variables, and runtime. Using an image ensures that the environment in your container is standardized and contains only what is necessary to build and run your application.

In this tutorial, you will create an application image for a static website that uses the Express framework and Bootstrap. You will then build a container using that image and push it to Docker Hub for future use. Finally, you will pull the stored image from your Docker Hub repository and build another container, demonstrating how you can recreate and scale your application.

Prerequisites

To follow this tutorial, you will need:

- One Ubuntu 18.04 server, set up following this Initial Server Setup guide.
- Docker installed on your server, following Steps 1 and 2 of How To Install and Use Docker on Ubuntu 18.04.
- Node.js and npm installed, following these instructions on installing with the PPA managed by NodeSource.
- A Docker Hub account. For an overview of how to set this up, refer to this introduction on getting started with Docker Hub.

Step 1 — Installing Your Application Dependencies

SCROLL TO TOP

To create your image, you will first need to make your application files, which you can then copy to your container. These files will include your application's static content, code, and dependencies.

First, create a directory for your project in your non-root user's home directory. We will call ours `node_project`, but you should feel free to replace this with something else:

```
$ mkdir node_project
```

Navigate to this directory:

```
$ cd node_project
```

This will be the root directory of the project.

Next, create a `package.json` file with your project's dependencies and other identifying information. Open the file with `nano` or your favorite editor:

```
$ nano package.json
```

Add the following information about the project, including its name, author, license, entrypoint, and dependencies. Be sure to replace the author information with your own name and contact details:

```
~/node_project/package.json

{
  "name": "nodejs-image-demo",
  "version": "1.0.0",
  "description": "nodejs image demo",
  "author": "Sammy the Shark <sammy@example.com>",
  "license": "MIT",
  "main": "app.js",
  "keywords": [
    "nodejs",
    "bootstrap",
    "express"
  ],
  "dependencies": {
    "express": "^4.16.4"
  }
}
```

This file includes the project name, author, and license under which it is being shared. Npm recommends making your project name short and descriptive, and avoiding duplicates in the npm registry. We've listed the MIT license in the license field, permitting the free use and distribution of the application code.

SCROLL TO TOP

Additionally, the file specifies:

- `"main"` : The entrypoint for the application, `app.js` . You will create this file next.
- `"dependencies"` : The project dependencies — in this case, Express 4.16.4 or above.

Though this file does not list a repository, you can add one by following these [guidelines on adding a repository to your `package.json` file](#). This is a good addition if you are versioning your application.

Save and close the file when you've finished making changes.

To install your project's dependencies, run the following command:

```
$ npm install
```

This will install the packages you've listed in your `package.json` file in your project directory.

We can now move on to building the application files.

Step 2 — Creating the Application Files

We will create a website that offers users information about sharks. Our application will have a main entrypoint, `app.js` , and a `views` directory that will include the project's static assets. The landing page, `index.html` , will offer users some preliminary information and a link to a page with more detailed shark information, `sharks.html` . In the `views` directory, we will create both the landing page and `sharks.html` .

First, open `app.js` in the main project directory to define the project's routes:

```
$ nano app.js
```

The first part of the file will create the Express application and Router objects, and define the base directory, port, and host as variables:

```
~/node_project/app.js

const express = require("express");
const app = express();
const router = express.Router();

const path = __dirname + '/views/';
const PORT = 8080;
const HOST = '0.0.0.0';
```

The `require` function loads the `express` module, which we then use to create the `app` and `router` objects. The `router` object will perform the routing function of the application, and as we define HTTP method routes we will add them to this object to define how our application will handle requests.

This section of the file also sets a few variables, `path` , `PORT` , and `HOST` :

SCROLL TO TOP

- `path` : Defines the base directory, which will be the `views` subdirectory within the current project directory.
- `HOST` : Defines the address that the application will bind to and listen on. Setting this to `0.0.0.0` or all IPv4 addresses corresponds with Docker's default behavior of exposing containers to `0.0.0.0` unless otherwise instructed.
- `PORT` : Tells the app to listen on and bind to port `8080`.

Next, set the routes for the application using the `router` object:

```
~/node_project/app.js
...

router.use(function (req,res,next) {
  console.log("/") + req.method);
  next();
});

router.get("/",function(req,res){
  res.sendFile(path + "index.html");
});

router.get("/sharks",function(req,res){
  res.sendFile(path + "sharks.html");
});
```

The `router.use` function loads a middleware function that will log the router's requests and pass them on to the application's routes. These are defined in the subsequent functions, which specify that a GET request to the base project URL should return the `index.html` page, while a GET request to the `/sharks` route should return `sharks.html`.

Finally, mount the `router` middleware and the application's static assets and tell the app to listen on port `8080`:

```
~/node_project/app.js
...

app.use(express.static(path));
app.use("/", router);

app.listen(8080, function () {
  console.log('Example app listening on port 8080!')
})
```

The finished `app.js` file will look like this:

```
~/node_project/app.js
```

SCROLL TO TOP

```

const express = require("express");
const app = express();
const router = express.Router();

const path = __dirname + '/views/';
const PORT = 8080;
const HOST = '0.0.0.0';

router.use(function (req,res,next) {
  console.log("/") + req.method);
  next();
});

router.get("/",function(req,res){
  res.sendFile(path + "index.html");
});

router.get("/sharks",function(req,res){
  res.sendFile(path + "sharks.html");
});

app.use(express.static(path));
app.use("/", router);

app.listen(8080, function () {
  console.log('Example app listening on port 8080!')
})

```

Save and close the file when you are finished.

Next, let's add some static content to the application. Start by creating the `views` directory:

```
$ mkdir views
```

Open the landing page file, `index.html` :

```
$ nano views/index.html
```

Add the following code to the file, which will import Bootstrap and create a jumbotron component with a link to the more detailed `sharks.html` info page:

```
~/node_project/views/index.html
```

```

<!DOCTYPE html>
<html lang="en">

```

```
<head>
```

```

<title>About Sharks</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.js">
<link href="css/styles.css" rel="stylesheet">
<link href="https://fonts.googleapis.com/css?family=Merriweather:400,700" rel="stylesheet" type="text/css">
</head>

<body>
  <nav class="navbar navbar-dark bg-dark navbar-static-top navbar-expand-md">
    <div class="container">
      <button type="button" class="navbar-toggler collapsed" data-toggle="collapse" data-target="#navbar-collapse">
        </button> <a class="navbar-brand" href="#">Everything Sharks</a>
      <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
        <ul class="nav navbar-nav mr-auto">
          <li class="active nav-item"><a href="/" class="nav-link">Home</a>
          </li>
          <li class="nav-item"><a href="/sharks" class="nav-link">Sharks</a>
          </li>
        </ul>
      </div>
    </div>
  </nav>

  <div class="jumbotron">
    <div class="container">
      <h1>Want to Learn About Sharks?</h1>
      <p>Are you ready to learn about sharks?</p>
      <br>
      <p><a class="btn btn-primary btn-lg" href="/sharks" role="button">Get Shark Info</a>
      </p>
    </div>
  </div>

  <div class="container">
    <div class="row">
      <div class="col-lg-6">
        <h3>Not all sharks are alike</h3>
        <p>Though some are dangerous, sharks generally do not attack humans. Out of the 500 species of sharks, only about 30 are known to be dangerous to humans.</p>
      </div>
      <div class="col-lg-6">
        <h3>Sharks are ancient</h3>
        <p>There is evidence to suggest that sharks lived up to 400 million years ago.</p>
      </div>
    </div>
  </div>
</body>

</html>

```

The top-level navbar here allows users to toggle between the **Home** and **Sharks** pages. In the `navbar-nav` subcomponent, we are using Bootstrap's `active` class to indicate the current page to the user. We've also specified the routes to our static pages, which match the routes we defined in `app.js`:

```
~/node_project/views/index.html

...
<div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
  <ul class="nav navbar-nav mr-auto">
    <li class="active nav-item"><a href="/" class="nav-link">Home</a>

    </li>
    <li class="nav-item"><a href="/sharks" class="nav-link">Sharks</a>
    </li>
  </ul>
</div>
...
```

Additionally, we've created a link to our shark information page in our jumbotron's button:

```
~/node_project/views/index.html

...
<div class="jumbotron">
  <div class="container">
    <h1>Want to Learn About Sharks?</h1>
    <p>Are you ready to learn about sharks?</p>
    <br>
    <p><a class="btn btn-primary btn-lg" href="/sharks" role="button">Get Shark Info</a>
    </p>
  </div>
</div>
...
```

There is also a link to a custom style sheet in the header:

```
~/node_project/views/index.html

...
<link href="css/styles.css" rel="stylesheet">
...
```

We will create this style sheet at the end of this step.

Save and close the file when you are finished.

With the application landing page in place, we can create our shark information page, `sharks.html`, which will offer interested users more information about sharks.

SCROLL TO TOP

Open the file:

```
$ nano views/sharks.html
```

Add the following code, which imports Bootstrap and the custom style sheet and offers users detailed information about certain sharks:

```
~/node_project/views/sharks.html
```

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>About Sharks</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css">
  <link href="css/styles.css" rel="stylesheet">
  <link href="https://fonts.googleapis.com/css?family=Merriweather:400,700" rel="stylesheet" type="text/css">
</head>
<nav class="navbar navbar-dark bg-dark navbar-static-top navbar-expand-md">
  <div class="container">
    <button type="button" class="navbar-toggler collapsed" data-toggle="collapse" data-target="#bs-example-navbar-collapse-1">
      </button> <a class="navbar-brand" href="/">Everything Sharks</a>
    <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
      <ul class="nav navbar-nav mr-auto">
        <li class="nav-item"><a href="/" class="nav-link">Home</a>
        </li>
        <li class="active nav-item"><a href="/sharks" class="nav-link">Sharks</a>
        </li>
      </ul>
    </div>
  </div>
</nav>
<div class="jumbotron text-center">
  <h1>Shark Info</h1>
</div>
<div class="container">
  <div class="row">
    <div class="col-lg-6">
      <p>
        <div class="caption">Some sharks are known to be dangerous to humans, though many more are friendly and welcoming!</div>
        
      </p>
    </div>
    <div class="col-lg-6">
      <p>
        <div class="caption">Other sharks are known to be friendly and welcoming!</div>
        
      </p>
    </div>
  </div>
</div>
```

SCROLL TO TOP


```
</div>
</div>
</div>

</html>
```

Note that in this file, we again use the `active` class to indicate the current page.

Save and close the file when you are finished.

Finally, create the custom CSS style sheet that you've linked to in `index.html` and `sharks.html` by first creating a `css` folder in the `views` directory:

```
$ mkdir views/css
```

Open the style sheet:

```
$ nano views/css/styles.css
```

Add the following code, which will set the desired color and font for our pages:

```
~/node_project/views/css/styles.css

.navbar {
  margin-bottom: 0;
}

body {
  background: #020A1B;
  color: #ffffff;
  font-family: 'Merriweather', sans-serif;
}

h1,
h2 {
  font-weight: bold;
}

p {
  font-size: 16px;
  color: #ffffff;
}

.jumbotron {
  background: #0048CD;
  color: white;
  text-align: center;
```

```

}

.jumbotron p {
  color: white;
  font-size: 26px;
}

.btn-primary {
  color: #fff;
  text-color: #000000;
  border-color: white;
  margin-bottom: 5px;
}

img,
video,
audio {
  margin-top: 20px;
  max-width: 80%;
}

div.caption: {
  float: left;
  clear: both;
}

```

In addition to setting font and color, this file also limits the size of the images by specifying a `max-width` of 80%. This will prevent them from taking up more room than we would like on the page.

Save and close the file when you are finished.

With the application files in place and the project dependencies installed, you are ready to start the application.

If you followed the initial server setup tutorial in the prerequisites, you will have an active firewall permitting only SSH traffic. To permit traffic to port `8080` run:

```
$ sudo ufw allow 8080
```

To start the application, make sure that you are in your project's root directory:

```
$ cd ~/node_project
```

Start the application with `node app.js`:

```
$ node app.js
```

Navigate your browser to `http://your_server_ip:8080` . You will see the following landing page:

Everything SharksHomeSharks

Want to Learn About Sharks?

Are you ready to learn about sharks?

Get Shark Info

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

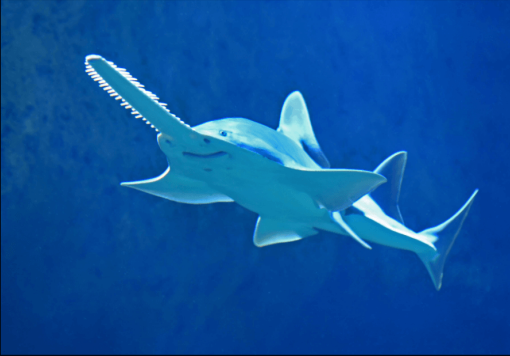
There is evidence to suggest that sharks lived up to 400 million years ago.

Click on the **Get Shark Info** button. You will see the following information page:

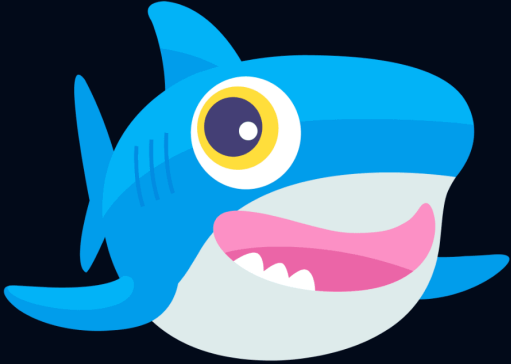
Everything SharksHomeSharks

Shark Info

Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.



Other sharks are known to be friendly and welcoming!



You now have an application up and running. When you are ready, quit the server by typing `CTRL+C` . We can now move on to creating the Dockerfile that will allow us to recreate and scale this application as desired.

Your Dockerfile specifies what will be included in your application container when it is executed. Using a Dockerfile allows you to define your container environment and avoid discrepancies with dependencies or runtime versions.

Following [these guidelines on building optimized containers](#), we will make our image as efficient as possible by minimizing the number of image layers and restricting the image's function to a single purpose — recreating our application files and static content.

In your project's root directory, create the Dockerfile:

```
$ nano Dockerfile
```

Docker images are created using a succession of layered images that build on one another. Our first step will be to add the *base image* for our application that will form the starting point of the application build.

Let's use the `node:10-alpine` image, since, at the time of writing, this is the [recommended LTS version of Node.js](#). The `alpine` image is derived from the [Alpine Linux project](#), and will help us keep our image size down. For more information about whether or not the `alpine` image is the right choice for your project, please see the full discussion under the **Image Variants** section of the [Docker Hub Node image page](#).

Add the following `FROM` instruction to set the application's base image:

```
~/node_project/Dockerfile
```

```
FROM node:10-alpine
```

This image includes Node.js and npm. Each Dockerfile must begin with a `FROM` instruction.

By default, the Docker Node image includes a non-root **node** user that you can use to avoid running your application container as **root**. It is a recommended security practice to avoid running containers as **root** and to [restrict capabilities within the container](#) to only those required to run its processes. We will therefore use the **node** user's home directory as the working directory for our application and set them as our user inside the container. For more information about best practices when working with the Docker Node image, see [this best practices guide](#).

To fine-tune the permissions on our application code in the container, let's create the `node_modules` subdirectory in `/home/node` along with the `app` directory. Creating these directories will ensure that they have the permissions we want, which will be important when we create local node modules in the container with `npm install`. In addition to creating these directories, we will set ownership on them to our **node** user:

```
~/node_project/Dockerfile
```

```
...
```

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

For more information on the utility of consolidating `RUN` instructions, see this [discussion of how to manage container layers](#).

Next, set the working directory of the application to `/home/node/app` :

```
~/node_project/Dockerfile
...
WORKDIR /home/node/app
```

If a `WORKDIR` isn't set, Docker will create one by default, so it's a good idea to set it explicitly.

Next, copy the `package.json` and `package-lock.json` (for npm 5+) files:

```
~/node_project/Dockerfile
...
COPY package*.json ./
```

Adding this `COPY` instruction before running `npm install` or copying the application code allows us to take advantage of Docker's caching mechanism. At each stage in the build, Docker will check to see if it has a layer cached for that particular instruction. If we change `package.json`, this layer will be rebuilt, but if we don't, this instruction will allow Docker to use the existing image layer and skip reinstalling our node modules.

After copying the project dependencies, we can run `npm install` :

```
~/node_project/Dockerfile
...
RUN npm install
```

Copy your application code to the working application directory on the container:

```
~/node_project/Dockerfile
...
COPY . .
```

To ensure that the application files are owned by the non-root **node** user, copy the permissions from your application directory to the directory on the container:

```
~/node_project/Dockerfile
...
COPY --chown=node:node . .
```

Set the user to **node**:

...

USER node

Expose port **8080** on the container and start the application:

~/node_project/Dockerfile

...

EXPOSE 8080

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

EXPOSE does not publish the port, but instead functions as a way of documenting which ports on the container will be published at runtime. **CMD** runs the command to start the application — in this case, `node app.js`. Note that there should only be one **CMD** instruction in each Dockerfile. If you include more than one, only the last will take effect.

There are many things you can do with the Dockerfile. For a complete list of instructions, please refer to Docker's [Dockerfile reference documentation](#).

The complete Dockerfile looks like this:

~/node_project/Dockerfile

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"]

Save and close the file when you are finished editing.

Before building the application image, let's add a `.dockerignore` file. Working in a similar way to a `.gitignore` file, `.dockerignore` specifies which files and directories in your project directory should not be copied over to your container.

Open the `.dockerignore` file:

```
$ nano .dockerignore
```

Inside the file, add your local node modules, npm logs, Dockerfile, and `.dockerignore` file:

```
~/node_project/.dockerignore

node_modules
npm-debug.log
Dockerfile
.dockerignore
```

If you are working with `Git` then you will also want to add your `.git` directory and `.gitignore` file.

Save and close the file when you are finished.

You are now ready to build the application image using the `docker build` command. Using the `-t` flag with `docker build` will allow you to tag the image with a memorable name. Because we are going to push the image to Docker Hub, let's include our Docker Hub username in the tag. We will tag the image as `nodejs-image-demo`, but feel free to replace this with a name of your own choosing. Remember to also replace `your_dockerhub_username` with your own Docker Hub username:

```
$ docker build -t your_dockerhub_username/nodejs-image-demo .
```

The `.` specifies that the build context is the current directory.

It will take a minute or two to build the image. Once it is complete, check your images:

```
$ docker images
```

You will see the following output:

Output

REPOSITORY	TAG	IMAGE ID	CREATED
your_dockerhub_username/nodejs-image-demo	latest	1c723fb2ef12	8 seconds
node	10-alpine	f09e7c96b6de	3 weeks ago

It is now possible to create a container with this image using `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 nodejs-image-demo -p 80:8080 -d your_dockerhub_username/nodejs-image-demo
```

Once your container is up and running, you can inspect a list of your running containers with `docker ps`:

```
$ docker ps
```

You will see the following output:

Output

CONTAINER ID	IMAGE	COMMAND	CREATED
e50ad27074a7	your_dockerhub_username/nodejs-image-demo	"node app.js"	8 seconds ago

With your container running, you can now visit your application by navigating your browser to `http://your_server_ip`. You will see your application landing page once again:

Want to Learn About Sharks?

Are you ready to learn about sharks?

Get Shark Info

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 created an image for your application, you can push it to Docker Hub for future use.

Step 4 — Using a Repository to Work with Images

By pushing your application image to a registry like Docker Hub, you make it available for subsequent use as you build and scale your containers. We will demonstrate how this works by pushing the application image to a repository and then using the image to recreate our container.

The first step to pushing the image is to log in to the Docker Hub account you created in the prerequisites:

```
$ docker login -u your_dockerhub_username -p your_dockerhub_password
```

Logging in this way will create a `~/.docker/config.json` file in your user's home directory with your Docker Hub credentials.

You can now push the application image to Docker Hub using the tag you created earlier, **your_dockerhub_username/nodejs-image-demo**:

```
$ docker push your_dockerhub_username/nodejs-image-demo
```

Let's test the utility of the image registry by destroying our current application container and image and rebuilding them with the image in our repository.

First, list your running containers:

```
$ docker ps
```

SCROLL TO TOP

You will see the following output:

Output

CONTAINER ID	IMAGE	COMMAND	CREATED
e50ad27074a7	your_dockerhub_username/nodejs-image-demo	"node app.js"	3 minutes ago

Using the CONTAINER ID listed in your output, stop the running application container. Be sure to replace the highlighted ID below with your own CONTAINER ID :

```
$ docker stop e50ad27074a7
```

List your all of your images with the -a flag:

```
$ docker images -a
```

You will see the following output with the name of your image, your_dockerhub_username/nodejs-image-demo , along with the node image and the other images from your build:

Output

REPOSITORY	TAG	IMAGE ID	CREATED
your_dockerhub_username/nodejs-image-demo	latest	1c723fb2ef12	7 minutes ago
<none>	<none>	2e3267d9ac02	4 minutes ago
<none>	<none>	8352b41730b9	4 minutes ago
<none>	<none>	5d58b92823cb	4 minutes ago
<none>	<none>	3f1e35d7062a	4 minutes ago
<none>	<none>	02176311e4d0	4 minutes ago
<none>	<none>	8e84b33edcda	4 minutes ago
<none>	<none>	6a5ed70f86f2	4 minutes ago
<none>	<none>	776b2637d3c1	4 minutes ago
node	10-alpine	f09e7c96b6de	3 weeks ago

Remove the stopped container and all of the images, including unused or dangling images, with the following command:

```
$ 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.

You have now removed both the container running your application image and the image. [SCROLL TO TOP](#)
information on removing Docker containers, images, and volumes, please see [How To Remove Docker](#)

Images, Containers, and Volumes.

With all of your images and containers deleted, you can now pull the application image from Docker Hub:

```
$ docker pull your_dockerhub_username/nodejs-image-demo
```

List your images once again:

```
$ docker images
```

You will see your application image:

Output

REPOSITORY	TAG	IMAGE ID	CREATED
your_dockerhub_username/nodejs-image-demo	latest	1c723fb2ef12	11 minutes ago

You can now rebuild your container using the command from Step 3:

```
$ docker run --name nodejs-image-demo -p 80:8080 -d your_dockerhub_username/nodejs-image-demo
```

List your running containers:

```
$ docker ps
```

Output

CONTAINER ID	IMAGE	COMMAND	CREATED
f6bc2f50dff6	your_dockerhub_username/nodejs-image-demo	"node app.js"	4 seconds ago

Visit http://your_server_ip once again to view your running application.

Conclusion

In this tutorial you created a static web application with Express and Bootstrap, as well as a Docker image for this application. You used this image to create a container and pushed the image to Docker Hub. From there, you were able to destroy your image and container and recreate them using your Docker Hub repository.

If you are interested in learning more about how to work with tools like Docker Compose and Docker Machine to create multi-container setups, you can look at the following guides:

- [How To Install Docker Compose on Ubuntu 18.04.](#)
- [How To Provision and Manage Remote Docker Hosts with Docker Machine on Ubuntu 18.04.](#)

For general tips on working with container data, see:

- [How To Share Data between Docker Containers.](#)
- [How To Share Data Between the Docker Container and the Host.](#)

If you are interested in other Docker-related topics, please see our complete library of [Docker tutorials](#).

By: Kathleen Juell

♡ Upvote (9)

📌 Subscribe

🔗 Share



We just made it easier for you to deploy faster.

[TRY FREE](#)

Related Tutorials

[How To Install YunoHost on Debian 9](#)

[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](#)

[How To Set Up a Private Docker Registry on Ubuntu 18.04](#)

[How To Secure a Containerized Node.js Application with Nginx, Let's Encrypt, and Docker Compose](#)

4 Comments

[SCROLL TO TOP](#)

Leave a comment...

Log In to Comment

^ [thiagopescado](#) December 6, 2018

0 Hi Kathleen. Great tutorial. As pointed out here one should run a node app like CMD ["node","app.js"]. Regards.

^ [katjuell](#) MOD December 6, 2018

0 Hi [@thiagopescado](#) — thanks so much for this! It's interesting — I was basing that decision on the process outlined in the official Node documentation, but it seems as though that doc and the best practices doc are not fully aligned on this particular issue. I really appreciate you pointing this out and I'm glad you enjoyed the tutorial.

^ [MU5741N3](#) December 10, 2018

0 Great!!!

^ [michaeljjshannon](#) January 25, 2019

0 Very good tutorial! I've just started an app development practice but in spite of this, I understood most of the guide.



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

SCROLL TO TOP



Copyright © 2019 DigitalOcean™ Inc.

[Community](#) [Tutorials](#) [Questions](#) [Projects](#) [Tags](#) [Newsletter](#) [RSS](#) 

[Distros & One-Click Apps](#) [Terms, Privacy, & Copyright](#) [Security](#) [Report a Bug](#) [Write for DOnations](#) [Shop](#)