Build your Node image

Estimated reading time: 14 minutes

Prerequisites

Work through the orientation and setup in Get started Part 1 (/get-started/) to understand Docker concepts.

Enable BuildKit

Before we start building images, ensure you have enabled BuildKit on your machine. BuildKit allows you to build Docker images efficiently. For more information, see Building images with BuildKit (/develop/develop-images/build_enhancements/).

BuildKit is enabled by default for all users on Docker Desktop. If you have installed Docker Desktop, you don't have to manually enable BuildKit. If you are running Docker on Linux, you can enable BuildKit either by using an environment variable or by making BuildKit the default setting.

To set the BuildKit environment variable when running the `docker build` command, run:

```
$ DOCKER_BUILDKIT=1 docker build .
```

To enable docker BuildKit by default, set daemon configuration in `/etc/docker/daemon.json` feature to `true` and restart the daemon. If the `daemon.json` file doesn't exist, create new file called `daemon.json` and then add the following to the file.

```
{
  "features":{"buildkit" : true}
}
```

Restart the Docker daemon.

Overview

Now that we have a good overview of containers and the Docker platform, let's take a look at building our first image. An image includes everything you need to run an application - the code or binary, runtime, dependencies, and any other file system objects required.

To complete this tutorial, you need the following:

- Node.js version 12.18 or later. Download Node.js (https://nodejs.org/en/)
- Docker running locally: Follow the instructions to download and install Docker (/desktop/).
- An IDE or a text editor to edit files. We recommend using Visual Studio Code.

Sample application

Let's create a simple Node.js application that we can use as our example. Create a directory in your local machine named `node-docker` and follow the steps below to create a simple REST API.

```
$ cd [path to your node-docker directory]
$ npm init -y
$ npm install ronin-server ronin-mocks
$ touch server.js
```

Now, let's add some code to handle our REST requests. We'll use a mock server so we can focus on Dockerizing the application.

Open this working directory in your IDE and add the following code into the `server.js` file.

```
const ronin = require('ronin-server')
const mocks = require('ronin-mocks')

const server = ronin.server()

server.use('/', mocks.server(server.Router(), false, true))
server.start()
```

The mocking server is called `Ronin.js` and will listen on port 8000 by default. You can make POST requests to the root (/) endpoint and any JSON structure you send to the server will be saved in memory. You can also send GET requests to the same endpoint and receive an array of JSON objects that you have previously POSTed.

Test the application

Let's start our application and make sure it's running properly. Open your terminal and navigate to your working directory you created.

```
$ node server.js
```

To test that the application is working properly, we'll first POST some JSON to the API and then make a GET request to see that the data has been saved. Open a new terminal and run the following curl commands:

```
$ curl --request POST \
  --url http://localhost:8000/test \
  --header 'content-type: application/json' \
  --data '{"msg": "testing" }'
{"code":"success","payload":[{"msg":"testing","id":"31f23305-f5d0-4b4f-a16f-6f4c8ec93cf1"

$ curl http://localhost:8000/test
{"code":"success","meta":{"total":1,"count":1},"payload":[{"msg":"testing","id":"31f23305
```

Switch back to the terminal where our server is running. You should now see the following requests in the server logs.

```
2020-XX-31T16:35:08:4260  INFO: POST /test
2020-XX-31T16:35:21:3560  INFO: GET /test
```

Great! We verified that the application works. At this stage, you've completed testing the server script locally.

Press `CTRL-c` from within the terminal session where the server is running to stop it.

```
2021-08-06T12:11:33:8930  INFO: POST /test
2021-08-06T12:11:41:5860  INFO: GET /test
^Cshutting down...
```

We will now continue to build and run the application in Docker.

Create a Dockerfile for Node.js

A Dockerfile is a text document that contains the instructions to assemble a Docker image. When we tell Docker to build our image by executing the `docker build` command, Docker reads these instructions, executes them, and creates a Docker image as a result.

Let's walk through the process of creating a Dockerfile for our application. In the root of your project, create a file named `Dockerfile` and open this file in your text editor.

> **What to name your Dockerfile?**
>
> The default filename to use for a Dockerfile is `Dockerfile` (without a file- extension). Using the default name allows you to run the `docker build` command without having to specify additional command flags.
>
> Some projects may need distinct Dockerfiles for specific purposes. A common convention is to name these `Dockerfile.<something>` or `<something>.Dockerfile`. Such Dockerfiles can then be used through the `--file` (or `-f` shorthand) option on the `docker build` command. Refer to the "Specify a Dockerfile" section (/engine/reference/commandline/build/#specify-a-dockerfile--f) in the `docker build` reference to learn about the `--file` option.
>
> We recommend using the default (`Dockerfile`) for your project's primary Dockerfile, which is what we'll use for most examples in this guide.

The first line to add to a Dockerfile is a `# syntax` parser directive (/engine/reference/builder/#syntax). While *optional*, this directive instructs the Docker builder what syntax to use when parsing the Dockerfile, and allows older Docker versions with BuildKit enabled to upgrade the parser before starting the build. Parser directives (/engine/reference/builder/#parser-directives) must appear before any other comment, whitespace, or Dockerfile instruction in your Dockerfile, and should be the first line in Dockerfiles.

```
# syntax=docker/dockerfile:1
```

We recommend using `docker/dockerfile:1`, which always points to the latest release of the version 1 syntax. BuildKit automatically checks for updates of the syntax before building, making sure you are using the most current version.

Next, we need to add a line in our Dockerfile that tells Docker what base image we would like to use for our application.

```
# syntax=docker/dockerfile:1

FROM node:12.18.1
```

Docker images can be inherited from other images. Therefore, instead of creating our own base image, we'll use the official Node.js image that already has all the tools and packages that we need to run a Node.js application. You can think of this in the same way you would think about class inheritance in object oriented programming. For example, if we were able to create Docker images in JavaScript, we might write something like the following.

```
class MyImage extends NodeBaseImage {}
```

This would create a class called `MyImage` that inherited functionality from the base class `NodeBaseImage`.

In the same way, when we use the `FROM` command, we tell Docker to include in our image all the functionality from the `node:12.18.1` image.

> **Note**
>
> If you want to learn more about creating your own base images, see Creating base images (/develop/develop-images/baseimages/).

The `NODE_ENV` environment variable specifies the environment in which an application is running (usually, development or production). One of the simplest things you can do to improve performance is to set `NODE_ENV` to `production`.

```
ENV NODE_ENV=production
```

To make things easier when running the rest of our commands, let's create a working directory. This instructs Docker to use this path as the default location for all subsequent commands. This way we do not have to type out full file paths but can use relative paths based on the working directory.

```
WORKDIR /app
```

Usually the very first thing you do once you've downloaded a project written in Node.js is to install npm packages. This ensures that your application has all its dependencies installed into the `node_modules` directory where the Node runtime will be able to find them.

Before we can run `npm install`, we need to get our `package.json` and `package-lock.json` files into our images. We use the `COPY` command to do this. The `COPY` command takes two parameters. The first parameter tells Docker what file(s) you would like to copy into the image. The second parameter tells Docker where you want that file(s) to be copied to. We'll copy the `package.json` and `package-lock.json` file into our working directory `/app`.

```
COPY ["package.json", "package-lock.json*", "./"]
```

Note that, rather than copying the entire working directory, we are only copying the package.json file. This allows us to take advantage of cached Docker layers. Once we have our files inside the image, we can use the `RUN` command to execute the command npm install. This works exactly the same as if we were running npm install locally on our machine, but this time these Node modules will be installed into the `node_modules` directory inside our image.

```
RUN npm install --production
```

At this point, we have an image that is based on node version 12.18.1 and we have installed our dependencies. The next thing we need to do is to add our source code into the image. We'll use the COPY command just like we did with our `package.json` files above.

```
COPY . .
```

The COPY command takes all the files located in the current directory and copies them into the image. Now, all we have to do is to tell Docker what command we want to run when our image is run inside of a container. We do this with the `CMD` command.

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

Here's the complete Dockerfile.

```
# syntax=docker/dockerfile:1

FROM node:12.18.1
ENV NODE_ENV=production

WORKDIR /app

COPY ["package.json", "package-lock.json*", "./"]

RUN npm install --production

COPY . .

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

Create a .dockerignore file

To use a file in the build context, the Dockerfile refers to the file specified in an instruction, for example, a COPY instruction. To increase the build's performance, exclude files and directories by adding a .dockerignore file to the context directory. To improve the context load time create a `.dockerignore` file and add `node_modules` directory in it.

```
node_modules
```

Build image

Now that we've created our Dockerfile, let's build our image. To do this, we use the `docker build` command. The `docker build` command builds Docker images from a Dockerfile and a "context". A build's context is the set of files located in the specified PATH or URL. The Docker build process can access any of the files located in the context.

The build command optionally takes a `--tag` flag. The tag is used to set the name of the image and an optional tag in the format `'name:tag'`. We'll leave off the optional "tag" for now to help simplify things. If you do not pass a tag, Docker will use "latest" as its default tag. You'll see this in the last line of the build output.

Let's build our first Docker image.

```
$ docker build --tag node-docker .

[+] Building 93.8s (11/11) FINISHED
 => [internal] load build definition from dockerfile
 => => transferring dockerfile: 617B
 => [internal] load .dockerignore
 ...
 => [2/5] WORKDIR /app
 => [3/5] COPY [package.json, package-lock.json*, ./]
 => [4/5] RUN npm install --production
 => [5/5] COPY . .
```

View local images

To see a list of images we have on our local machine, we have two options. One is to use the CLI and the other is to use Docker Desktop. Since we are currently working in the terminal let's take a look at listing images with the CLI.

To list images, simply run the `images` command.

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED              SIZE
node-docker         latest              3809733582bc        About a minute ago   945MB
```

Your exact output may vary, but you should see the image we just built `node-docker:latest` with the `latest` tag.

Tag images

An image name is made up of slash-separated name components. Name components may contain lowercase letters, digits and separators. A separator is defined as a period, one or two underscores, or one or more dashes. A name component may not start or end with a separator.

An image is made up of a manifest and a list of layers. In simple terms, a “tag” points to a combination of these artifacts. You can have multiple tags for an image. Let’s create a second tag for the image we built and take a look at its layers.

To create a new tag for the image we built above, run the following command.

```
$ docker tag node-docker:latest node-docker:v1.0.0
```

The Docker tag command creates a new tag for an image. It does not create a new image. The tag points to the same image and is just another way to reference the image.

Now run the `docker images` command to see a list of our local images.

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
node-docker         latest              3809733582bc        24 minutes ago      945MB
node-docker         v1.0.0              3809733582bc        24 minutes ago      945MB
```

You can see that we have two images that start with `node-docker`. We know they are the same image because if you look at the IMAGE ID column, you can see that the values are the same for the two images.

Let’s remove the tag that we just created. To do this, we’ll use the rmi command. The rmi command stands for “remove image”.

```
$ docker rmi node-docker:v1.0.0
Untagged: node-docker:v1.0.0
```

Notice that the response from Docker tells us that the image has not been removed but only “untagged”. Verify this by running the images command.

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
node-docker         latest              3809733582bc        32 minutes ago      945MB
```

Our image that was tagged with `:v1.0.0` has been removed but we still have the `node-docker:latest` tag available on our machine.

Next steps

In this module, we took a look at setting up our example Node application that we will use for the rest of the tutorial. We also created a Dockerfile that we used to build our Docker image. Then, we took a look at tagging our images and removing images. In the next module, we’ll take a look at how

to:

Run your image as a container (/language/nodejs/run-containers/)

Feedback

Help us improve this topic by providing your feedback. Let us know what you think by creating an issue in the Docker Docs (https://github.com/docker/docker.github.io/issues/new?title=[Node.js%20docs%20feedback]) GitHub repository. Alternatively, create a PR (https://github.com/docker/docker.github.io/pulls) to suggest updates.

containers (/search/?q=containers), images (/search/?q=images), node.js (/search/?q=node.js), node (/search/?q=node), dockerfiles (/search/?q=dockerfiles), node (/search/?q=node), coding (/search/?q=coding), build (/search/?q=build), push (/search/?q=push), run (/search/?q=run)