Docker allows you to package an application with all of its dependencies into a standardized unit, called a container, for software development. A container is a stripped-to-basics version of a Linux operating system. An image is software you load into a container.

## Create the Node.js app

First, create a new directory where all the files would live. In this directory create a package.json file that describes your app and its dependencies:

{

"name": "docker\_web\_app",

"version": "1.0.0",

"description": "Node.js on Docker",

"author": "First Last <first.last@example.com>",

"main": "server.js",

"scripts": {

"start": "node server.js"

},

"dependencies": {

"express": "^4.16.1"

}

}

With your new package.json file, run npm install. If you are using npm version 5 or later, this will generate a package-lock.json file which will be copied to your Docker image.

Then, create a server.js file that defines a web app using the [Express.js](https://expressjs.com/) framework:

'use strict';

const express = require('express');

// Constants

const PORT = 8080;

const HOST = '0.0.0.0';

// App

const app = express();

app.get('/', (req, res) => {

res.send('Hello world\n');

});

app.listen(PORT, HOST);

console.log(`Running on http://${HOST}:${PORT}`);

In the next steps, we'll look at how you can run this app inside a Docker container using the official Docker image. First, you'll need to build a Docker image of your app.

## Creating a Dockerfile

Create an empty file called Dockerfile:

touch Dockerfile

Open the Dockerfile in your favorite text editor

The first thing we need to do is define from what image we want to build from. Here we will use the latest LTS (long term support) version 8 of node available from the [Docker Hub](https://hub.docker.com/):

FROM node:8

Next we create a directory to hold the application code inside the image, this will be the working directory for your application:

# Create app directory

WORKDIR /usr/src/app

This image comes with Node.js and NPM already installed so the next thing we need to do is to install your app dependencies using the npm binary. Please note that if you are using npm version 4 or earlier a package-lock.json file will not be generated.

# Install app dependencies

# A wildcard is used to ensure both package.json AND package-lock.json are copied

# where available (npm@5+)

COPY package\*.json ./

RUN npm install

# If you are building your code for production

# RUN npm install --only=production

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. bitJudo has a good explanation of this [here](http://bitjudo.com/blog/2014/03/13/building-efficient-dockerfiles-node-dot-js/).

To bundle your app's source code inside the Docker image, use the COPY instruction:

# Bundle app source

COPY . .

Your app binds to port 8080 so you'll use the EXPOSE instruction to have it mapped by the docker daemon:

EXPOSE 8080

Last but not least, define the command to run your app using CMD which defines your runtime. Here we will use the basic npm start which will run node server.js to start your server:

CMD [ "npm", "start" ]

Your Dockerfile should now look like this:

FROM node:8

# Create app directory

WORKDIR /usr/src/app

# Install app dependencies

# A wildcard is used to ensure both package.json AND package-lock.json are copied

# where available (npm@5+)

COPY package\*.json ./

RUN npm install

# If you are building your code for production

# RUN npm install --only=production

# Bundle app source

COPY . .

EXPOSE 8080

CMD [ "npm", "start" ]

## .dockerignore file

Create a .dockerignore file in the same directory as your Dockerfile with following content:

node\_modules

npm-debug.log

This will prevent your local modules and debug logs from being copied onto your Docker image and possibly overwriting modules installed within your image.

## Building your image

Go to the directory that has your Dockerfile and run the following command to build the Docker image. The -t flag lets you tag your image so it's easier to find later using the docker images command:

$ docker build -t <your username>/node-web-app .

Your image will now be listed by Docker:

$ docker images

# Example

REPOSITORY TAG ID CREATED

node 8 1934b0b038d1 5 days ago

<your username>/node-web-app latest d64d3505b0d2 1 minute ago

## Run the image

Running your image with -d runs the container in detached mode, leaving the container running in the background. The -p flag redirects a public port to a private port inside the container. Run the image you previously built:

$ docker run -p 49160:8080 -d <your username>/node-web-app

Print the output of your app:

# Get container ID

$ docker ps

# Print app output

$ docker logs <container id>

# Example

Running on http://localhost:8080

If you need to go inside the container you can use the exec command:

# Enter the container

$ docker exec -it <container id> /bin/bash

## Test

To test your app, get the port of your app that Docker mapped:

$ docker ps

# Example

ID IMAGE COMMAND ... PORTS

ecce33b30ebf <your username>/node-web-app:latest npm start ... 49160->8080

In the example above, Docker mapped the 8080 port inside of the container to the port 49160 on your machine.

Now you can call your app using curl (install if needed via: sudo apt-get install curl):

$ curl -i localhost:49160

HTTP/1.1 200 OK

X-Powered-By: Express

Content-Type: text/html; charset=utf-8

Content-Length: 12

ETag: W/"c-M6tWOb/Y57lesdjQuHeB1P/qTV0"

Date: Mon, 13 Nov 2017 20:53:59 GMT

Connection: keep-alive

Hello world

We hope this tutorial helped you get up and running a simple Node.js application on Docker.

|  |
| --- |
| **Pushing Docker image** |
|  |

|  |
| --- |
| export DOCKER\_ID\_USER="username" |
|  |

|  |
| --- |
| docker login |
|  |

|  |
| --- |
| (find image id by : sudo docker images) |
|  |

|  |
| --- |
| sudo docker tag 6d0fa3d26dcb andalike/node-web-app |
|  |

sudo docker push andalike/node-web-app

SERVICES In Docker:

Save this file as docker-compose.yml wherever you want.

version: "3"

services:

web:

# replace username/repo:tag with your name and image details

image: andalike/node-web-app

deploy:

replicas: 5

resources:

limits:

cpus: "0.1"

memory: 50M

restart\_policy:

condition: on-failure

ports:

- "4000:80"

networks:

- webnet

networks:

webnet:

## Run your new load-balanced app

Before we can use the docker stack deploy command we first run:

docker swarm init

Now let’s run it. You need to give your app a name. Here, it is set to getstartedlab:

docker stack deploy -c docker-compose.yml getstartedlab

Our single service stack is running 5 container instances of our deployed image on one host. Let’s investigate.

Get the service ID for the one service in our application:

docker service ls

Look for output for the web service, prepended with your app name. If you named it the same as shown in this example, the name is getstartedlab\_web. The service ID is listed as well, along with the number of replicas, image name, and exposed ports.

A single container running in a service is called a **task**. Tasks are given unique IDs that numerically increment, up to the number of replicas you defined in docker-compose.yml. List the tasks for your service:

docker service ls

docker service ps getstartedlab\_web

Tasks also show up if you just list all the containers on your system, though that is not filtered by service:

docker container ls -q

## Scale the app

You can scale the app by changing the replicas value in docker-compose.yml, saving the change, and re-running the docker stack deploy command:

docker stack deploy -c docker-compose.yml getstartedlab

Docker performs an in-place update, no need to tear the stack down first or kill any containers.

Now, re-run docker container ls -q to see the deployed instances reconfigured. If you scaled up the replicas, more tasks, and hence, more containers, are started.

### Take down the app and the swarm

* Take the app down with docker stack rm:
* docker stack rm getstartedlab
* Take down the swarm.
* docker swarm leave --force

**Python with Docker**

## Your new development environment

In the past, if you were to start writing a Python app, your first order of business was to install a Python runtime onto your machine. But, that creates a situation where the environment on your machine needs to be perfect for your app to run as expected, and also needs to match your production environment.

With Docker, you can just grab a portable Python runtime as an image, no installation necessary. Then, your build can include the base Python image right alongside your app code, ensuring that your app, its dependencies, and the runtime, all travel together.

These portable images are defined by something called a Dockerfile.

## Define a container with Dockerfile

Dockerfile defines what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you need to map ports to the outside world, and be specific about what files you want to “copy in” to that environment. However, after doing that, you can expect that the build of your app defined in this Dockerfile behaves exactly the same wherever it runs.

### Dockerfile

Create an empty directory. Change directories (cd) into the new directory, create a file called Dockerfile, copy-and-paste the following content into that file, and save it. Take note of the comments that explain each statement in your new Dockerfile.

# Use an official Python runtime as a parent image

FROM python:2.7-slim

# Set the working directory to /app

WORKDIR /app

# Copy the current directory contents into the container at /app

COPY . /app

# Install any needed packages specified in requirements.txt

RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container

EXPOSE 80

# Define environment variable

ENV NAME World

# Run app.py when the container launches

CMD ["python", "app.py"]

This Dockerfile refers to a couple of files we haven’t created yet, namely app.py and requirements.txt. Let’s create those next.

## The app itself

Create two more files, requirements.txt and app.py, and put them in the same folder with the Dockerfile. This completes our app, which as you can see is quite simple. When the above Dockerfile is built into an image, app.py and requirements.txt is present because of that Dockerfile’s COPY command, and the output from app.py is accessible over HTTP thanks to the EXPOSE command.

### requirements.txt

Flask

Redis

### app.py

from flask import Flask

from redis import Redis, RedisError

import os

import socket

# Connect to Redis

redis = Redis(host="redis", db=0, socket\_connect\_timeout=2, socket\_timeout=2)

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

try:

visits = redis.incr("counter")

except RedisError:

visits = "<i>cannot connect to Redis, counter disabled</i>"

html = "<h3>Hello {name}!</h3>" \

"<b>Hostname:</b> {hostname}<br/>" \

"<b>Visits:</b> {visits}"

return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host='0.0.0.0', port=80)

Now we see that pip install -r requirements.txt installs the Flask and Redis libraries for Python, and the app prints the environment variable NAME, as well as the output of a call to socket.gethostname(). Finally, because Redis isn’t running (as we’ve only installed the Python library, and not Redis itself), we should expect that the attempt to use it here fails and produces the error message.

**Note**: Accessing the name of the host when inside a container retrieves the container ID, which is like the process ID for a running executable.

That’s it! You don’t need Python or anything in requirements.txt on your system, nor does building or running this image install them on your system. It doesn’t seem like you’ve really set up an environment with Python and Flask, but you have.

## Build the app

We are ready to build the app. Make sure you are still at the top level of your new directory. Here’s what ls should show:

$ ls

Dockerfile app.py requirements.txt

Now run the build command. This creates a Docker image, which we’re going to tag using -t so it has a friendly name.

docker build -t friendlyhello .

Where is your built image? It’s in your machine’s local Docker image registry:

$ docker image ls

REPOSITORY TAG IMAGE ID

friendlyhello latest 326387cea398

Troubleshooting for Linux users

Proxy server settings

Proxy servers can block connections to your web app once it’s up and running. If you are behind a proxy server, add the following lines to your Dockerfile, using the ENV command to specify the host and port for your proxy servers:

# Set proxy server, replace host:port with values for your servers

ENV http\_proxy host:port

ENV https\_proxy host:port

DNS settings

DNS misconfigurations can generate problems with pip. You need to set your own DNS server address to make pip work properly. You might want to change the DNS settings of the Docker daemon. You can edit (or create) the configuration file at /etc/docker/daemon.json with the dns key, as following:

{

"dns": ["your\_dns\_address", "8.8.8.8"]

}

In the example above, the first element of the list is the address of your DNS server. The second item is the Google’s DNS which can be used when the first one is not available.

Before proceeding, save daemon.json and restart the docker service.

sudo service docker restart

Once fixed, retry to run the build command.

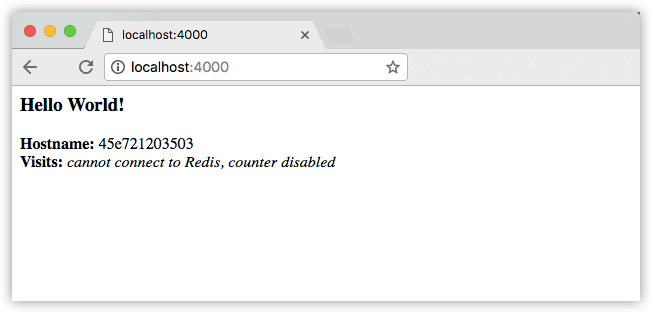
## Run the app

Run the app, mapping your machine’s port 4000 to the container’s published port 80 using -p:

docker run -p 4000:80 friendlyhello

You should see a message that Python is serving your app at http://0.0.0.0:80. But that message is coming from inside the container, which doesn’t know you mapped port 80 of that container to 4000, making the correct URL http://localhost:4000.

Go to that URL in a web browser to see the display content served up on a web page.



**Note**: If you are using Docker Toolbox on Windows 7, use the Docker Machine IP instead of localhost. For example, http://192.168.99.100:4000/. To find the IP address, use the command docker-machine ip.

You can also use the curl command in a shell to view the same content.

$ curl http://localhost:4000

<h3>Hello World!</h3><b>Hostname:</b> 8fc990912a14<br/><b>Visits:</b> <i>cannot connect to Redis, counter disabled</i>

This port remapping of 4000:80 demonstrates the difference between EXPOSE within the Dockerfile and what the publish value is set to when running docker run -p. In later steps, map port 4000 on the host to port 80 in the container and use http://localhost.

Hit CTRL+C in your terminal to quit.

On Windows, explicitly stop the container

On Windows systems, CTRL+C does not stop the container. So, first type CTRL+C to get the prompt back (or open another shell), then type docker container ls to list the running containers, followed by docker container stop <Container NAME or ID> to stop the container. Otherwise, you get an error response from the daemon when you try to re-run the container in the next step.

Now let’s run the app in the background, in detached mode:

docker run -d -p 4000:80 friendlyhello

You get the long container ID for your app and then are kicked back to your terminal. Your container is running in the background. You can also see the abbreviated container ID with docker container ls (and both work interchangeably when running commands):

$ docker container ls

CONTAINER ID IMAGE COMMAND CREATED

1fa4ab2cf395 friendlyhello "python app.py" 28 seconds ago

Notice that CONTAINER ID matches what’s on http://localhost:4000.

Now use docker container stop to end the process, using the CONTAINER ID, like so:

docker container stop 1fa4ab2cf395

## Share your image

To demonstrate the portability of what we just created, let’s upload our built image and run it somewhere else. After all, you need to know how to push to registries when you want to deploy containers to production.

A registry is a collection of repositories, and a repository is a collection of images—sort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The docker CLI uses Docker’s public registry by default.

### Log in with your Docker ID

If you don’t have a Docker account, sign up for one at [hub.docker.com](https://hub.docker.com). Make note of your username.

Log in to the Docker public registry on your local machine.

$ docker login

### Tag the image

The notation for associating a local image with a repository on a registry is username/repository:tag. The tag is optional, but recommended, since it is the mechanism that registries use to give Docker images a version. Give the repository and tag meaningful names for the context, such as get-started:part2. This puts the image in the get-started repository and tag it as part2.

Now, put it all together to tag the image. Run docker tag image with your username, repository, and tag names so that the image uploads to your desired destination. The syntax of the command is:

docker tag image username/repository:tag

For example:

sudo docker tag 6edf906dcc81 andalike/friendlyhello

Run [docker image ls](https://docs.docker.com/engine/reference/commandline/image_ls/) to see your newly tagged image.

$ docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

friendlyhello latest 6edf906dcc81 3 minutes ago 195MB

### Publish the image

Upload your tagged image to the repository:

docker push username/repository:tag

Once complete, the results of this upload are publicly available. If you log in to [Docker Hub](https://hub.docker.com/), you see the new image there, with its pull command.

### Pull and run the image from the remote repository

From now on, you can use docker run and run your app on any machine with this command:

docker run -p 4000:80 andalike/friendlyhello

If the image isn’t available locally on the machine, Docker pulls it from the repository.

ubuntu@ip-172-31-36-164:~$ sudo docker run -p 4000:80 andalike/friendlyhello

\* Serving Flask app "app" (lazy loading)

\* Environment: production

WARNING: Do not use the development server in a production environment.

Use a production WSGI server instead.

\* Debug mode: off

\* Running on http://0.0.0.0:80/ (Press CTRL+C to quit)

No matter where docker run executes, it pulls your image, along with Python and all the dependencies from requirements.txt, and runs your code. It all travels together in a neat little package, and you don’t need to install anything on the host machine for Docker to run it.

Here is a list of the basic Docker commands from this page, and some related ones if you’d like to explore a bit before moving on.

docker build -t friendlyhello . # Create image using this directory's Dockerfile

docker run -p 4000:80 friendlyhello # Run "friendlyname" mapping port 4000 to 80

docker run -d -p 4000:80 friendlyhello # Same thing, but in detached mode

docker container ls # List all running containers

docker container ls -a # List all containers, even those not running

docker container stop <hash> # Gracefully stop the specified container

docker container kill <hash> # Force shutdown of the specified container

docker container rm <hash> # Remove specified container from this machine

docker container rm $(docker container ls -a -q) # Remove all containers

docker image ls -a # List all images on this machine

docker image rm <image id> # Remove specified image from this machine

docker image rm $(docker image ls -a -q) # Remove all images from this machine

docker login # Log in this CLI session using your Docker credentials

docker tag <image> username/repository:tag # Tag <image> for upload to registry

docker push username/repository:tag # Upload tagged image to registry

docker run username/repository:tag # Run image from a registry