Ben Scuron

Professor Williams

CIS 4360

15 February 2023

Lab 2 - Containers

Table of Contents

[1. Lab 2 - Containers](#org4883f2e)

[1.1. Hello](#org2e33870)

[1.2. Build](#orga66ab0d)

[1.3. Run](#org8075de9)

[1.4. Debug](#orgcfcf0d2)

[1.5. Publish](#org681f6e8)

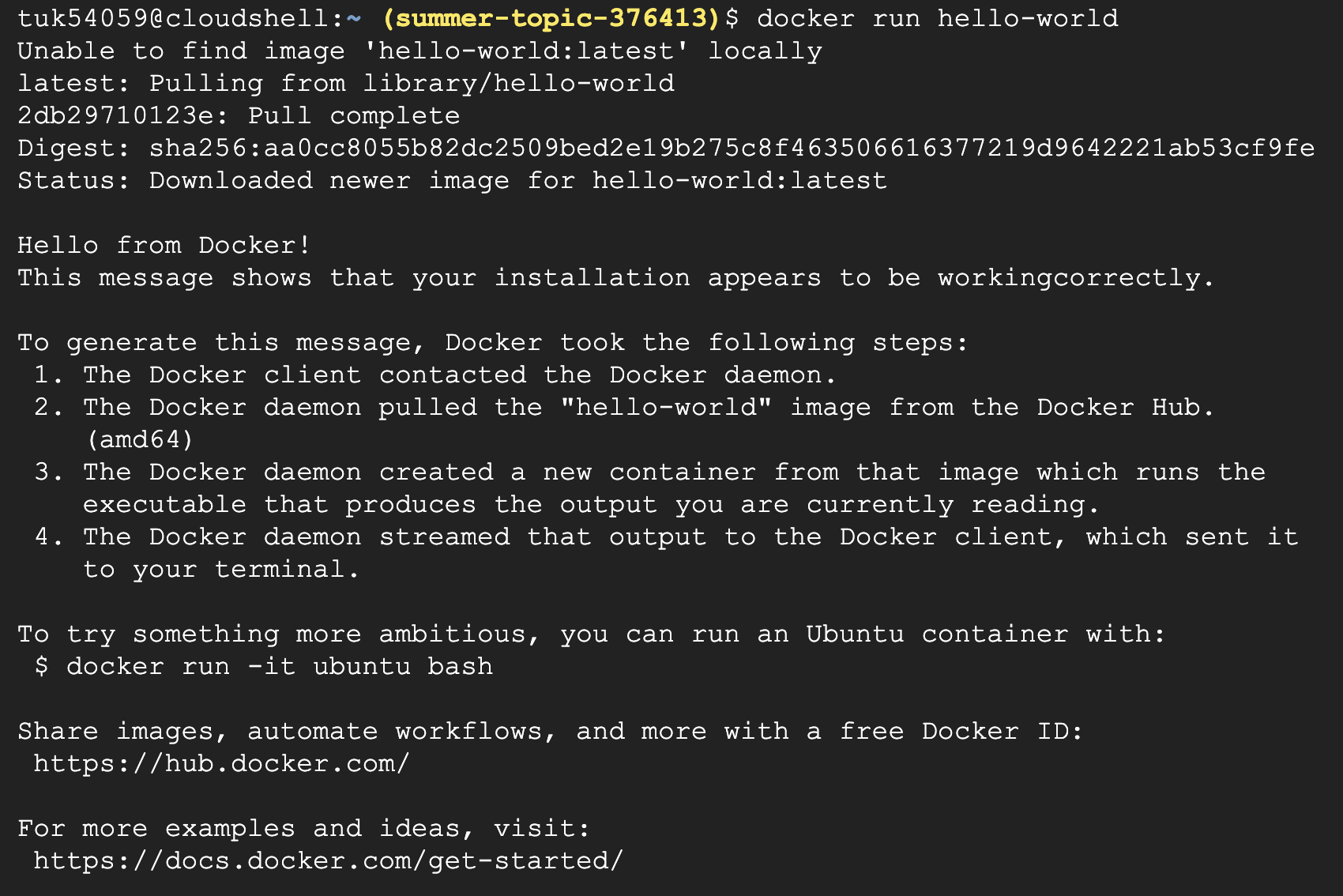
# Lab 2 - Containers

## Hello

First, I made sure that Docker was installed on my cloud instance:

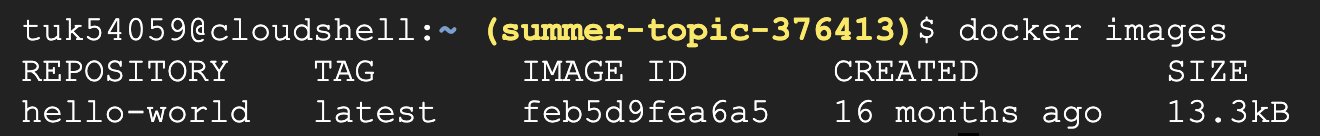


Next, I ran the command to run a hello world Docker container:



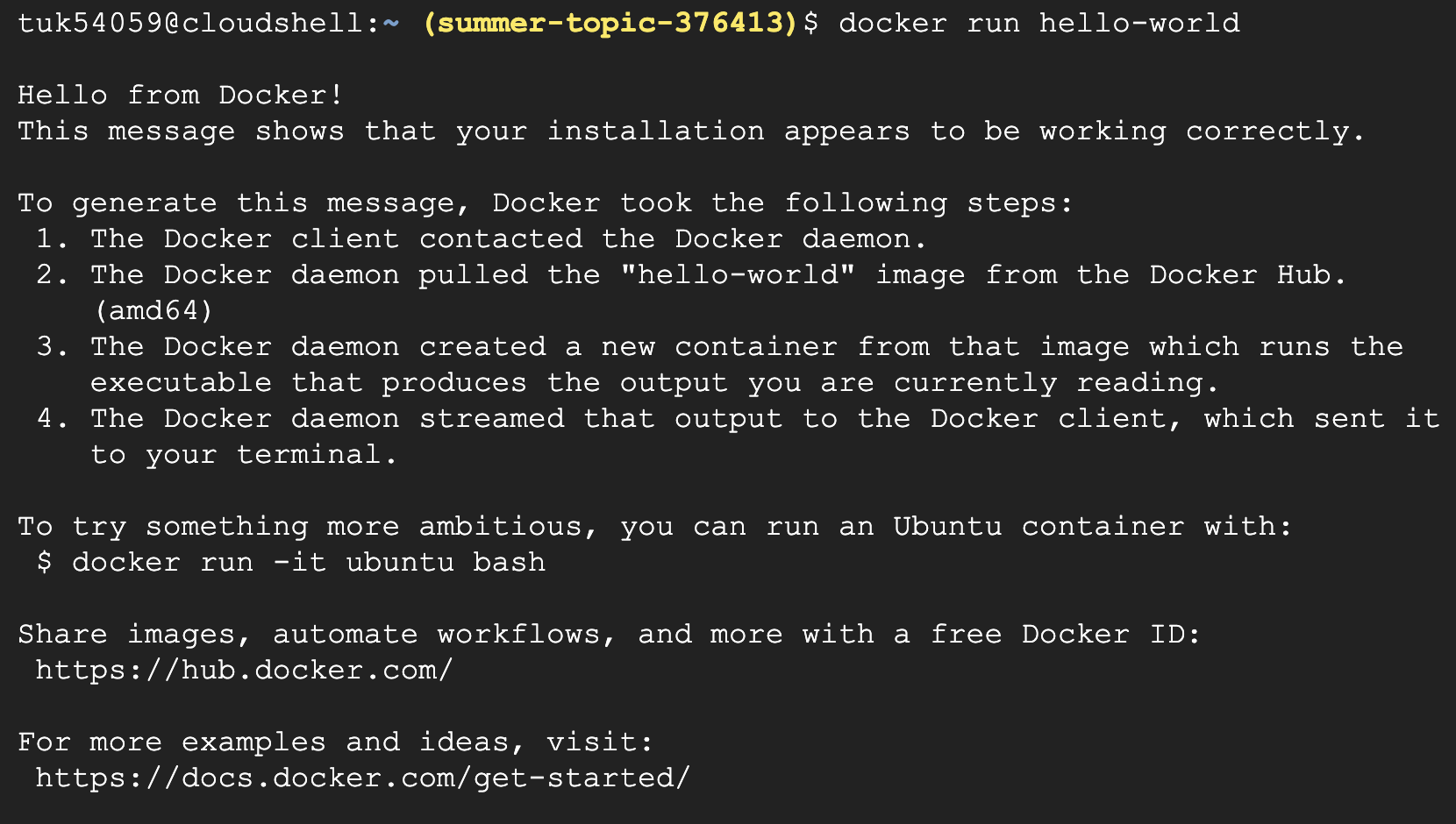
The image was not installed on my cloud machine, so it was pulled from a remote registry known as the "Docker Hub."

I then ran a command that lists my current Docker images:



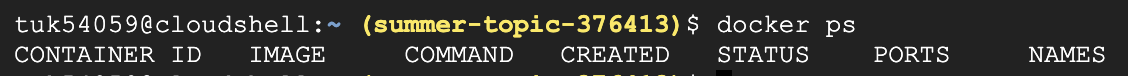
As you can see from the output above, the image that was pulled from the Docker Hub is 16 months old, has the tag latest (which means it is the latest version), has the Image ID "feb5d9fea6a5", and is 13.3 kilobytes (kB) in size. I was interested to see where exactly the image is written to the cloud machine's drive. I did some research and found that the images are stored in the directory "/var/lib/docker." Unfortunately, I could not access that directory due to lack of permission on the cloud console.

Now, I will try to run the same hello world container. This time, it should not need to pull from the Docker Hub, since the container is stored locally:

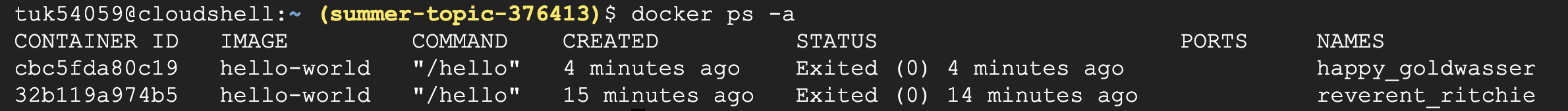


As you can see, Docker was able to find the image locally, without having to pull it down remotely from the Docker Hub registry.

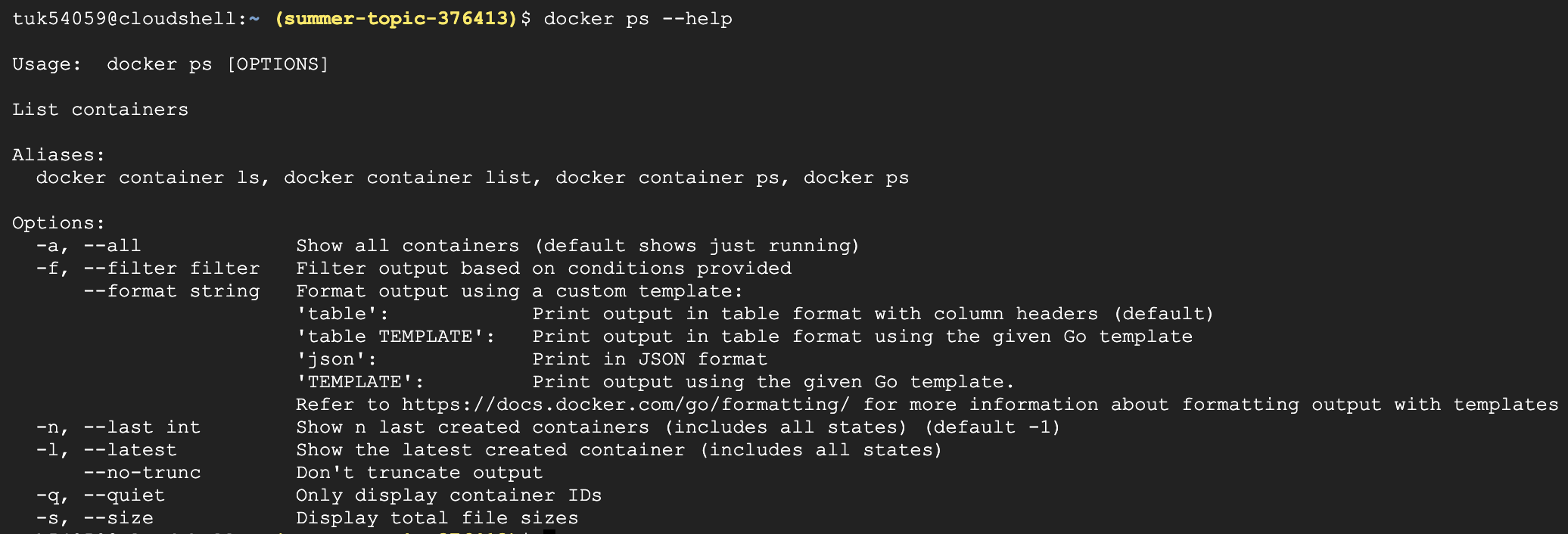
To check if there are Docker containers currently running, I ran the following command:



Since there are no running containers currently, no containers are listed in the output. Now, I want to check all Docker containers, not just the running containers. This output should show the previous "hello-world" containers that I ran:



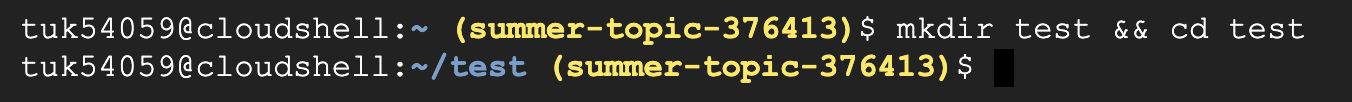
I was interested to see what exactly the -a flag did when appended to the docker ps command. To do this, I first ran the "docker help" command. At the bottom of this command, there was a message about how to learn more about specific Docker commands: "Run 'docker COMMAND –help' for more information on a command." Since I wanted to learn more about the docker ps command I typed: "docker ps –help":



After running this command, more information is printed about the docker ps command. Specifically, I was able to figure out what exactly the -a flag was doing. The -a flag is used to specify: "Show all containers (default shows just running)."

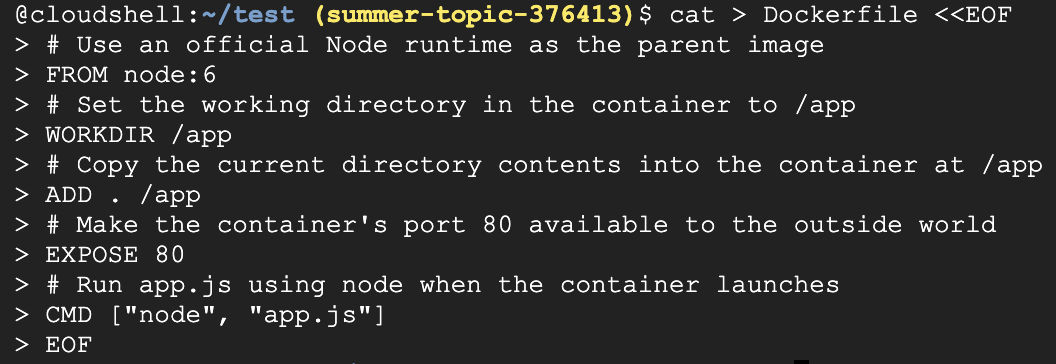
## Build

In the build section, we are going to create a Docker image. The first thing we will do is create a directory and change our current directory to that newly created directory:

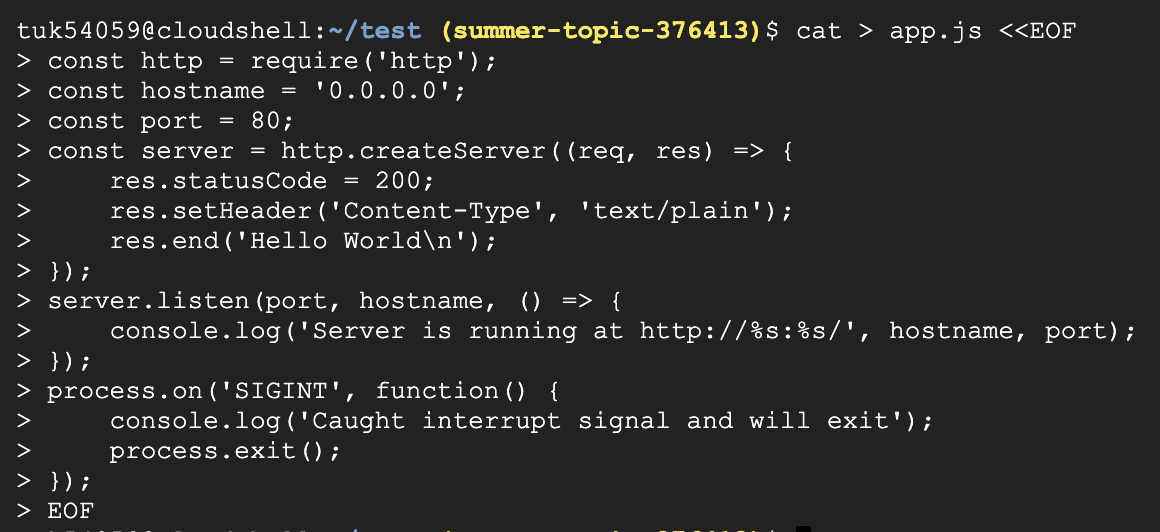


The two ampersands between the mkdir command and cd command is a way to run the second command only if the first command succeeded.

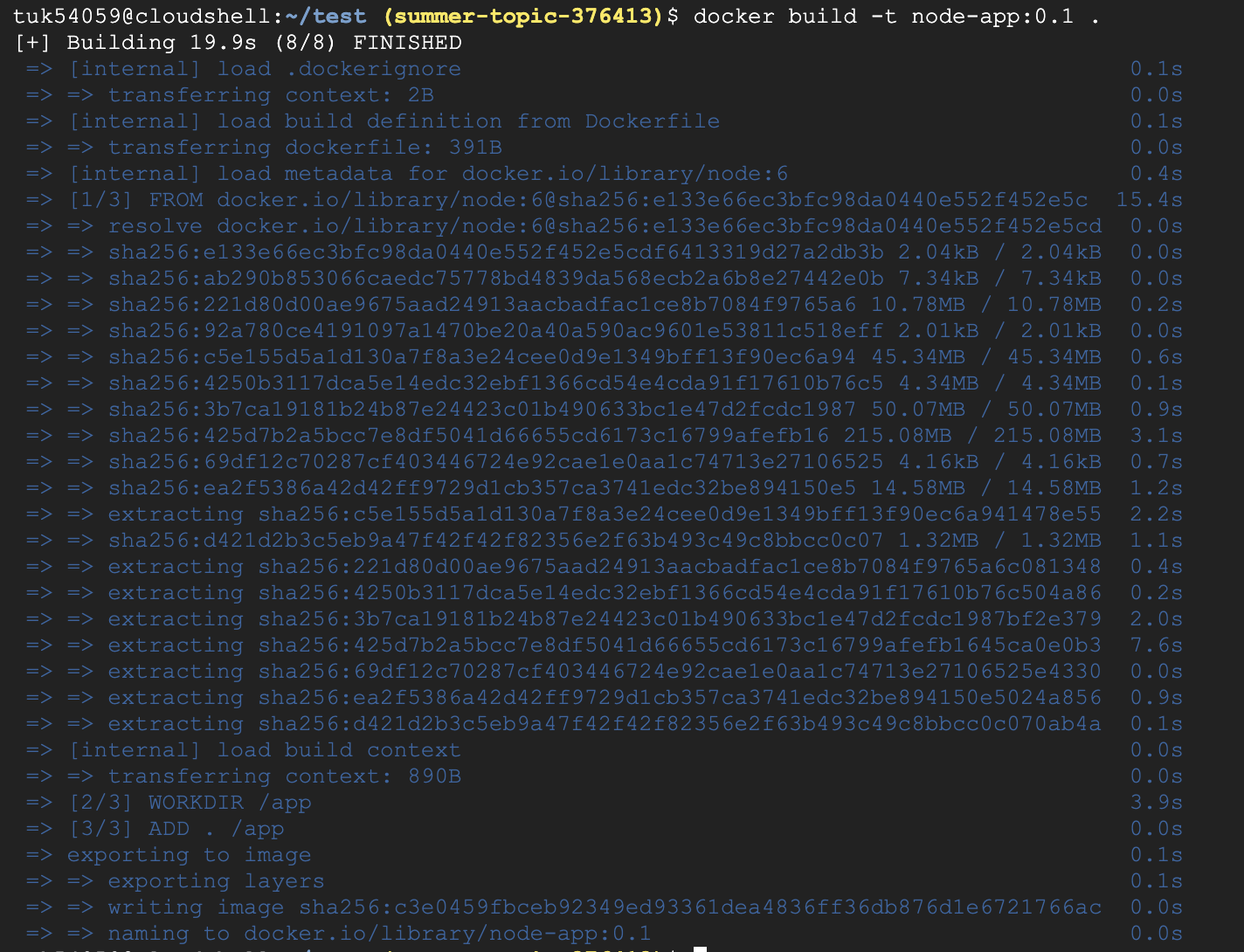
Next, a Dockerfile is created. Dockerfiles are used to specify how to build the image we want to create. In the Dockerfile that we create, the first line will specify the Node version that we are using (version 6). The next line will set the working directory to "/app." Next, we will copy the contents of the current directory into the working directory "/app." After that, we will expose port 80 to others. Lastly, we will specify the command to run when the container is run. In this case, we specify the command to run is "node app.js." The Dockerfile can be seen below:



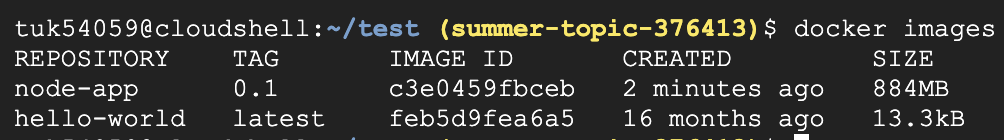
Next, I created a node HTTP server that will listen on port 80 and return "Hello World" to users. To do this I created a file named app.js and inserted the following JavaScript code to setup the web server:



Now we are ready to build the Docker image. To do so, I ran the following command:



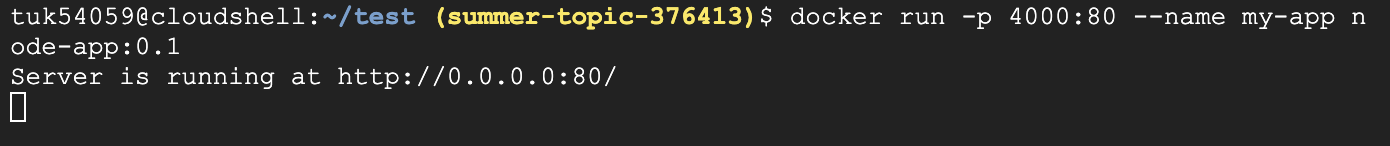
As you can see, the image was built successfully. Just to double check, I ran the command below:



In the output of the command above, our node-app image that was just created is visible.

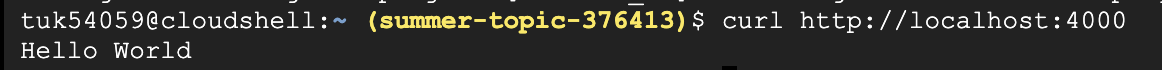
## Run

Now that we have successfully built the Docker image, it is time to run it. Remember, the web server will run on port 80.



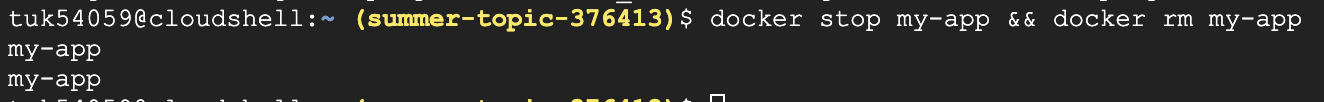
The web server is now running on port 80. The 0.0.0.0 IP address means all IPv4 addresses on the local machine.

Now to test if I can connect to the web server, I ran the curl command in another console instance:

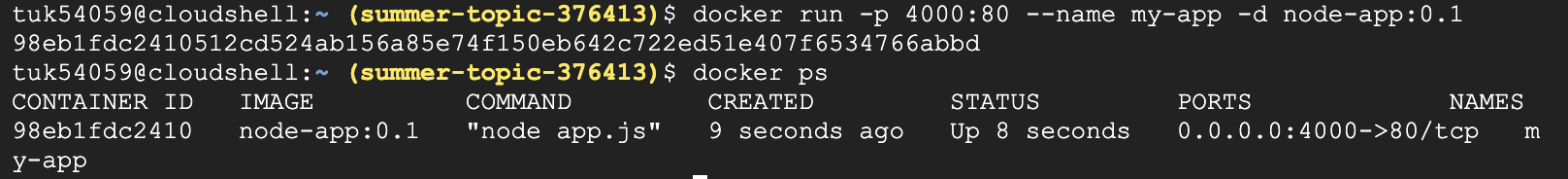


After connecting to our web server that we created, I received the Hello World message that we were expecting.

Next, I ran the following command to stop and delete the container that we created:



Now, I started the container again, but this time I ran it as a background process and used docker ps to check that it was running in the background:

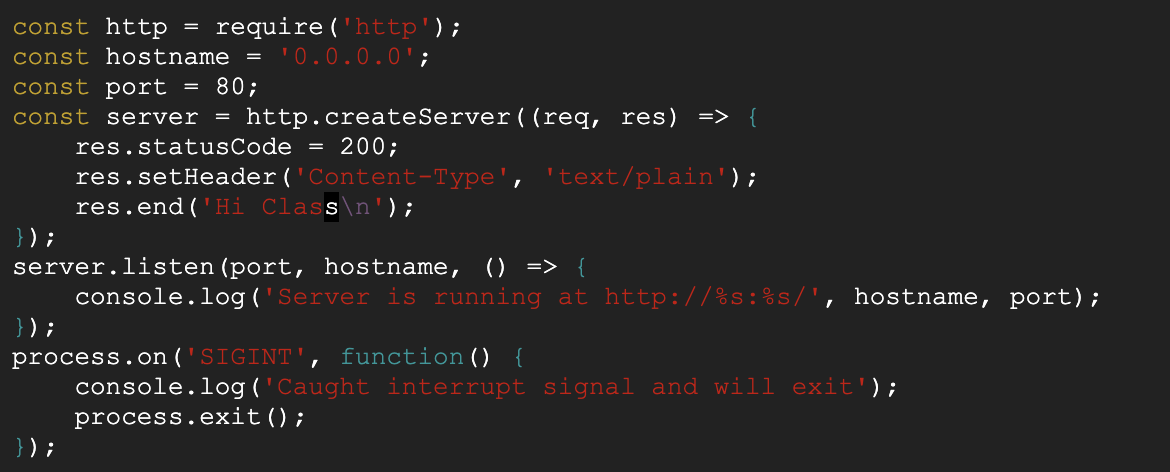


To check the output of a container running in the background, we can use the docker logs command. To see the logs of a specific container, we pass the container id to the docker logs command:

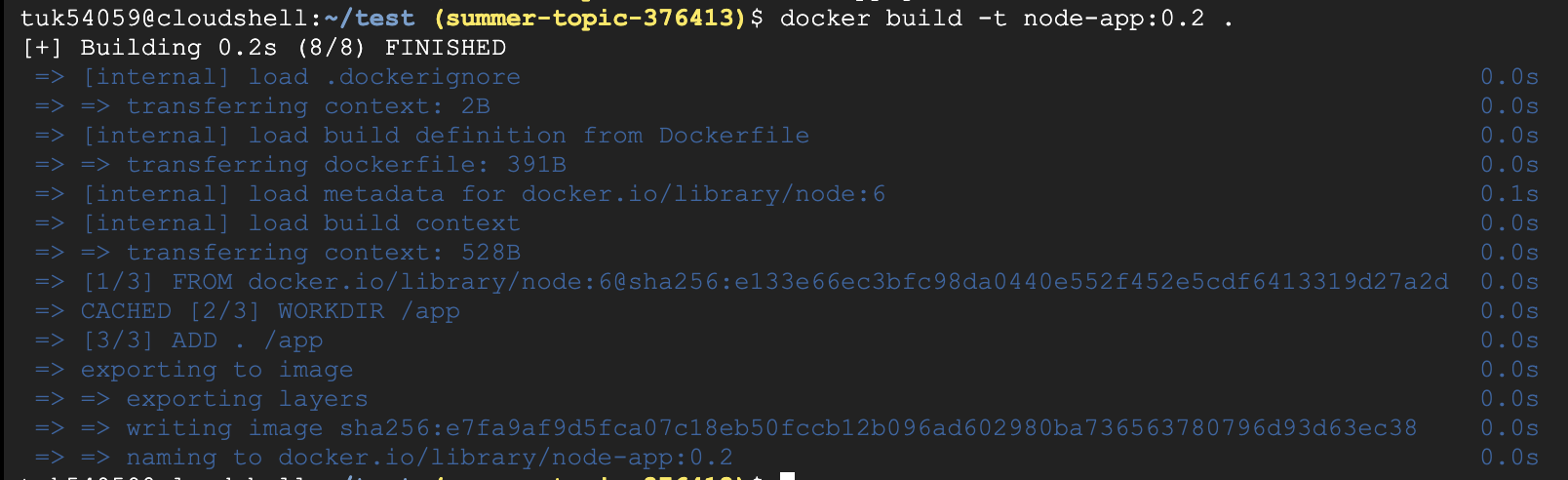


As you can see from the output of Docker logs, the container running our web server is running on port 80 just as expected.

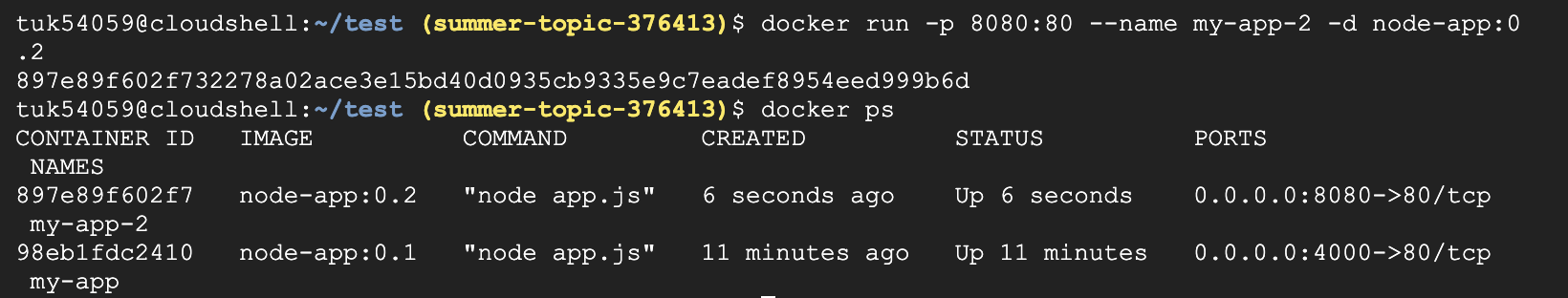
Now, I will change the text "Hello World" in our application to some other text. To do this, I used the vim text editor. First, I opened our app.js file in vim. Next, I searched for the text "Hello" by pressing '/' and continued to spell out Hello. Next, I pressed 'ct\' to delete until the '\' character and get placed in insert mode. I then typed my new text, "Hi Class". I then pressed escape to return to normal mode, and then pressed ':wq' to write and quit. Below is a screenshot of my updated app.js file inside of vim:



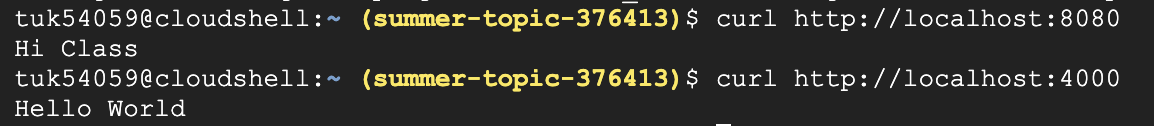
I now rebuild the updated image with a new tag (0.2):



It was built successfully. I then ran the newly created container and checked that it was running alongside the old container:



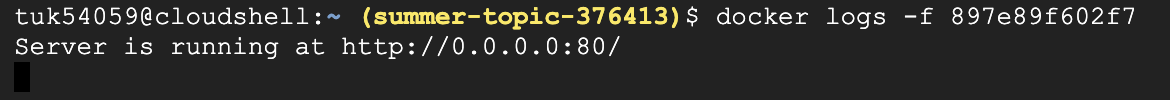
As you can see from the screenshot above, both of our web applications are running. I then tested the newly created container to make sure that the output matches the new text that I added to app.js, while the old container still shows Hello World:



I think this is really useful how we can have two containers running at the same time, listening on different ports. It was also very easy to send each process to run in the background, while still being able to easily interact with them. The setup for these containers was also very trivial and it was easy to get up and going in a short amount of time.

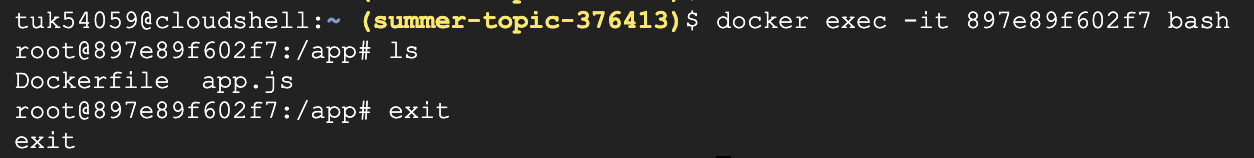
## Debug

In this section, I try to familiarize myself with debugging Docker containers. One useful command is the docker logs command that we used previously:



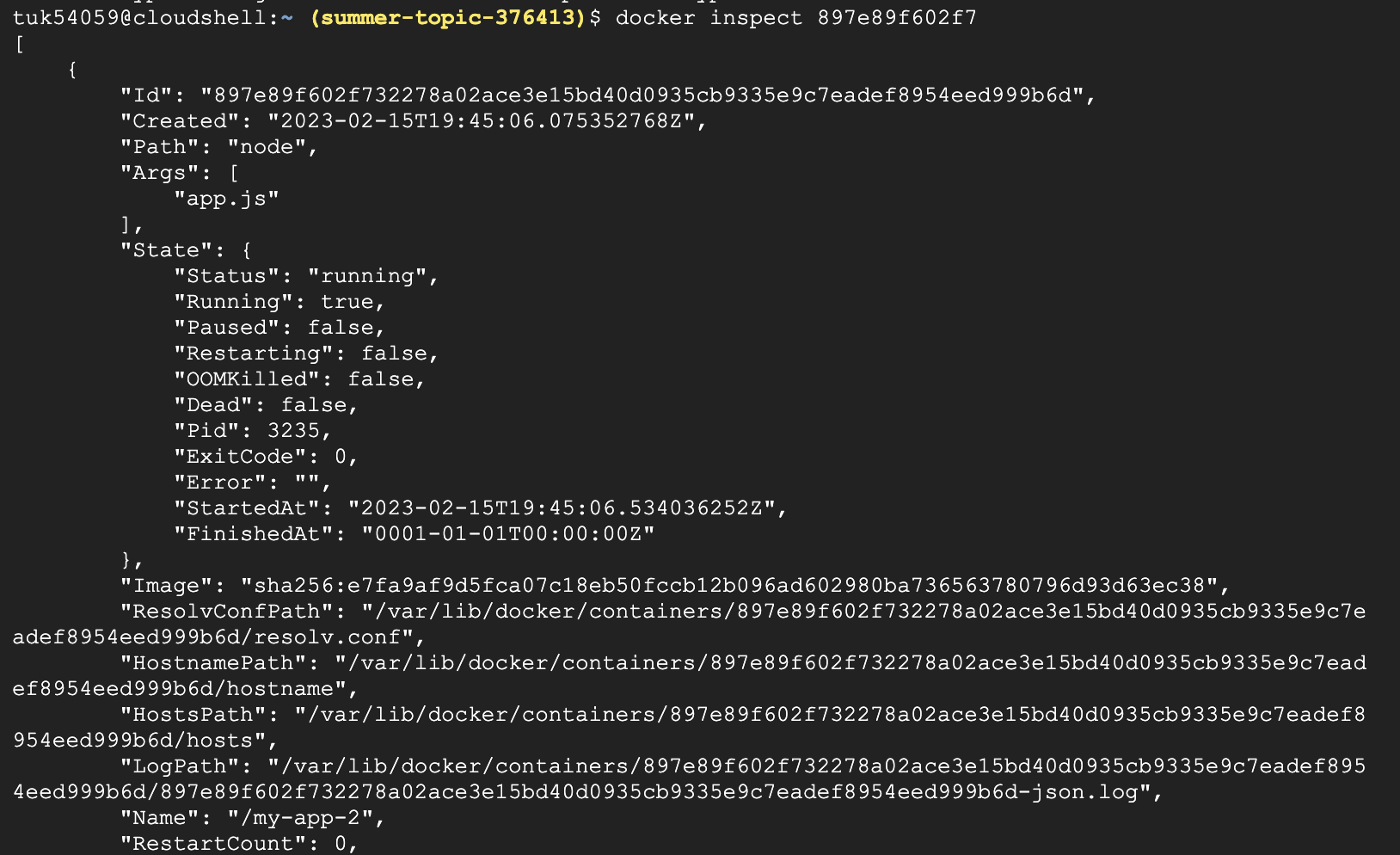
From the output above, we can tell that our newer web server is running on port 80.

Sometimes it may be useful to create an interactive Bash shell for a Docker container. To do this, I ran the following command:

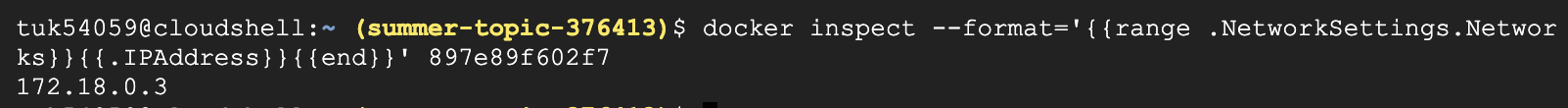


This is useful for when you need to run a series of commands in a Docker container. This interactive shell allows you to easily navigate within the Docker container itself.

Next, I used the docker inspect command to obtain low-level information on my specific Docker container:



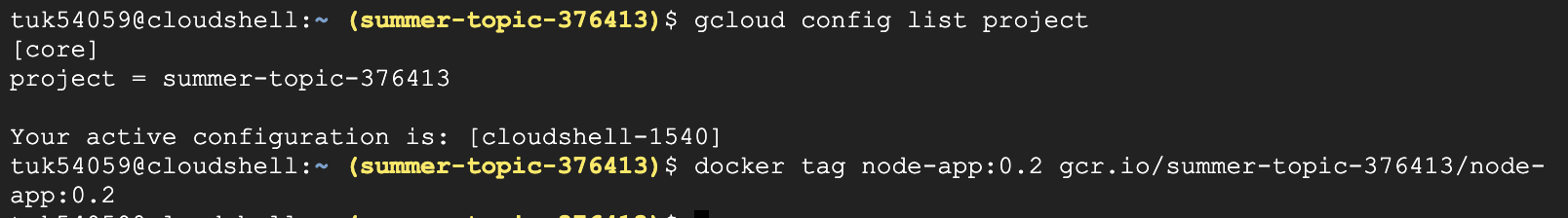
The output of this command does not fit onto one screenshot. The output is formatted in JSON. To inspect a specific field from the output, we can set the format flag:



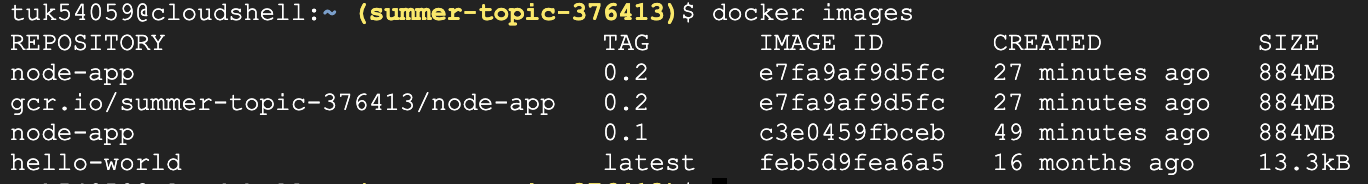
In the command above, I inspected a specific field to obtain the IP address from the JSON output (172.18.0.3).

## Publish

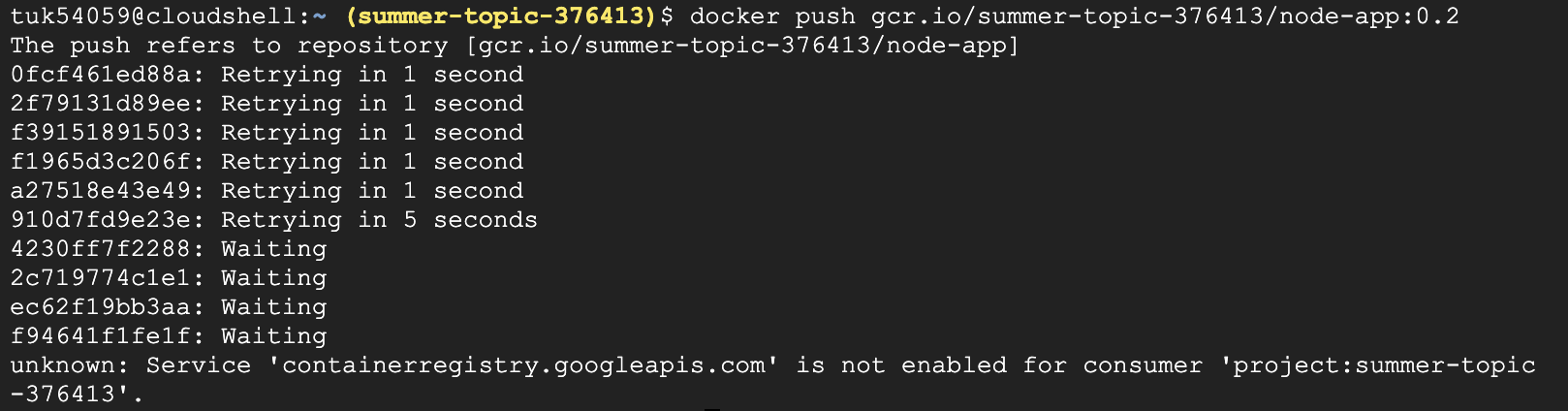
Lastly, we will publish our image to the google container registry. I used my project ID (summer-topic-376413) and configured it to use my newer node application created before:



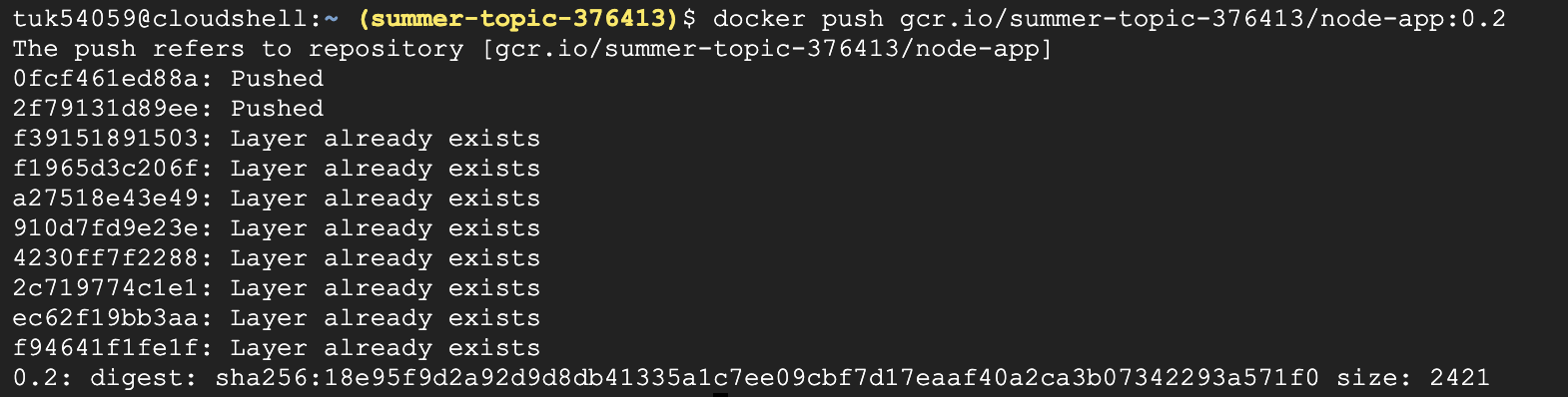
I then checked to make sure that my newly tagged image was visible using docker images:



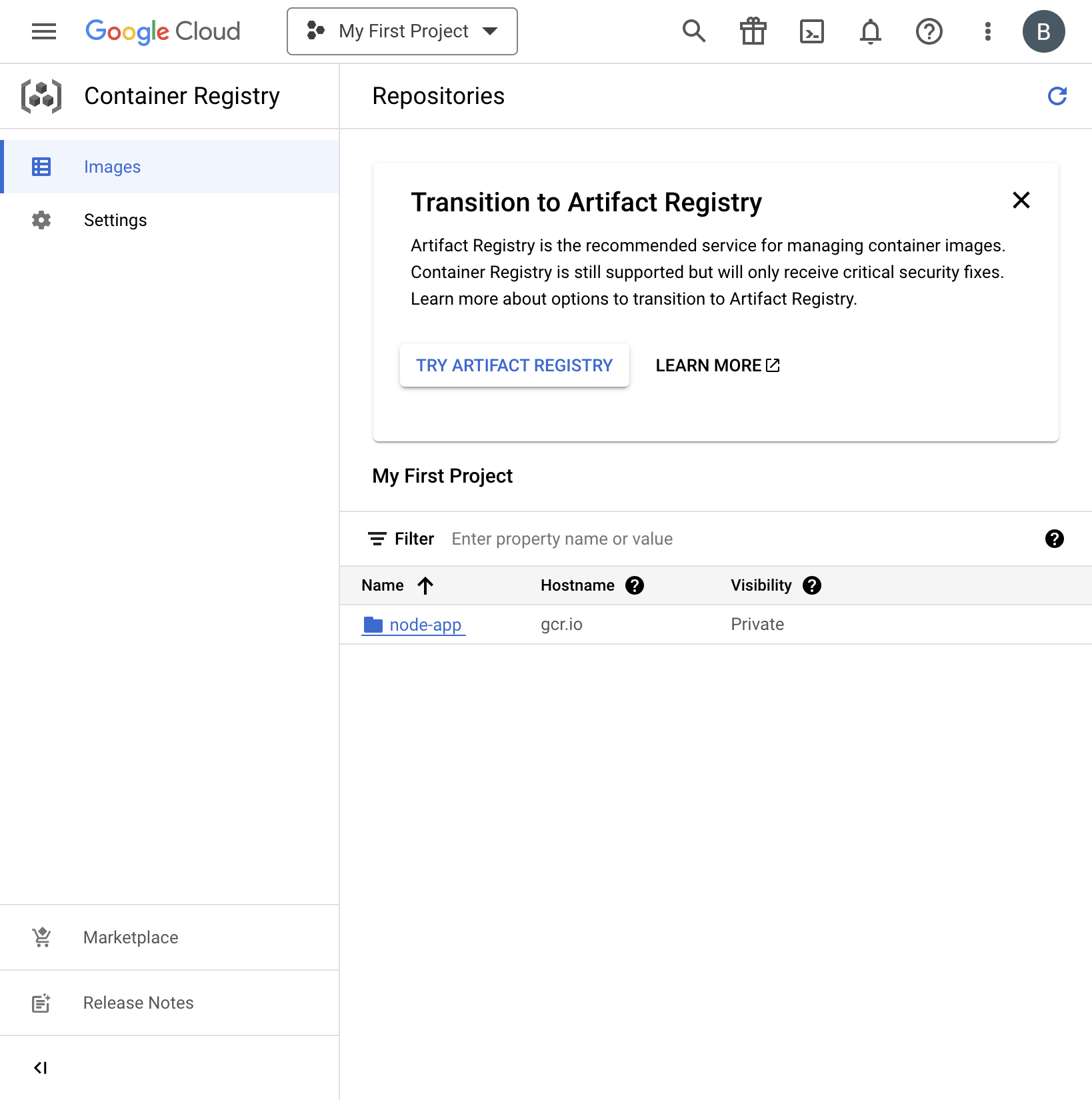
Next, I finally tried to publish the image:



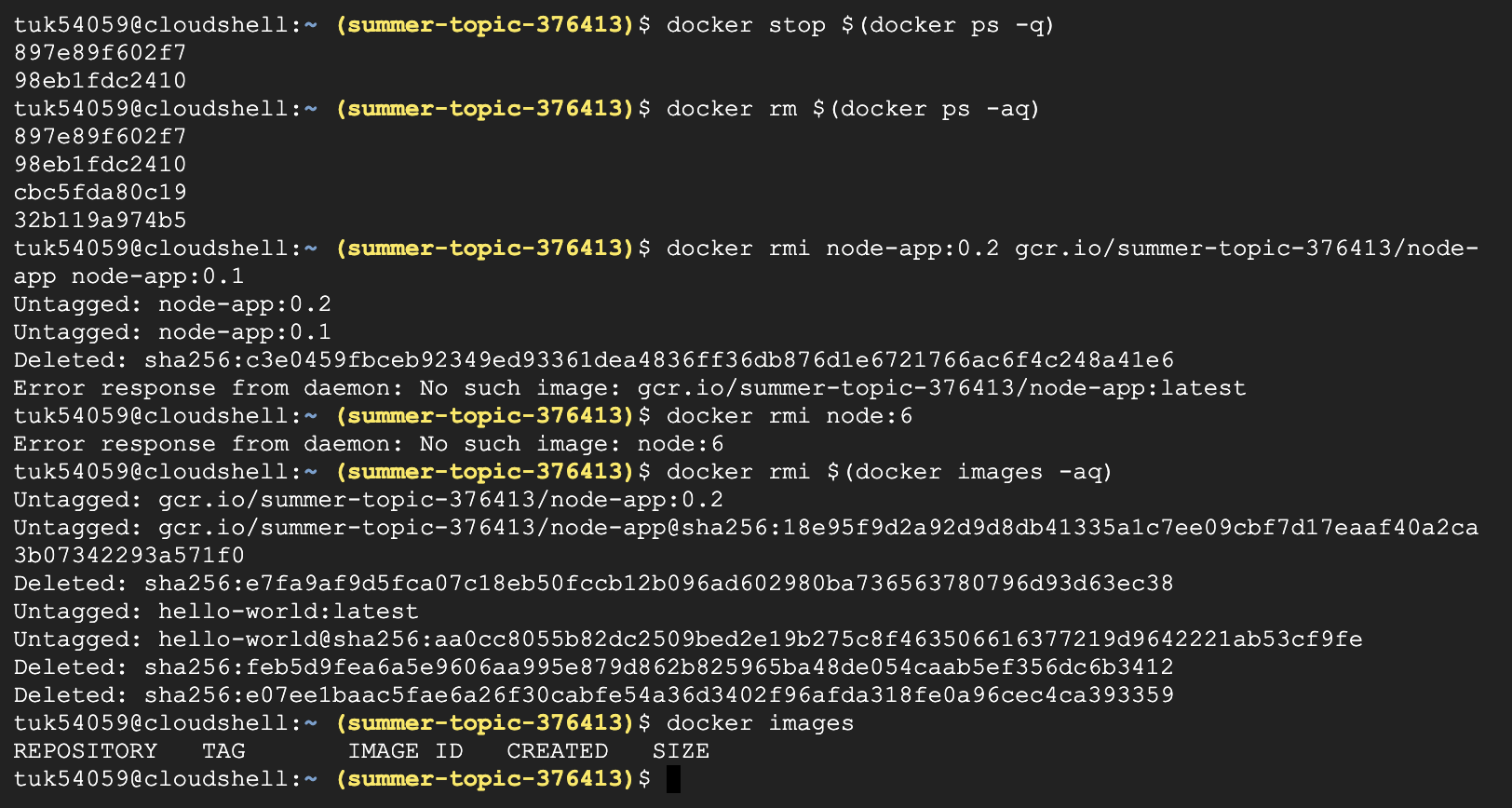
My first push initially failed, since I did not have the container registry service enabled. After enabling that service within the GCP, I was able to publish the container:



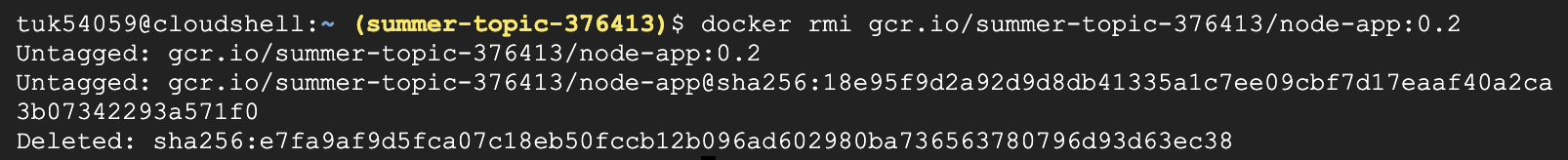
After publishing, my container was visible within the Google Cloud Container Registry:



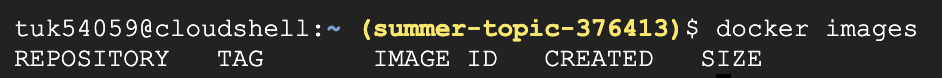
Next, it was time to cleanup the containers on our local cloud console:



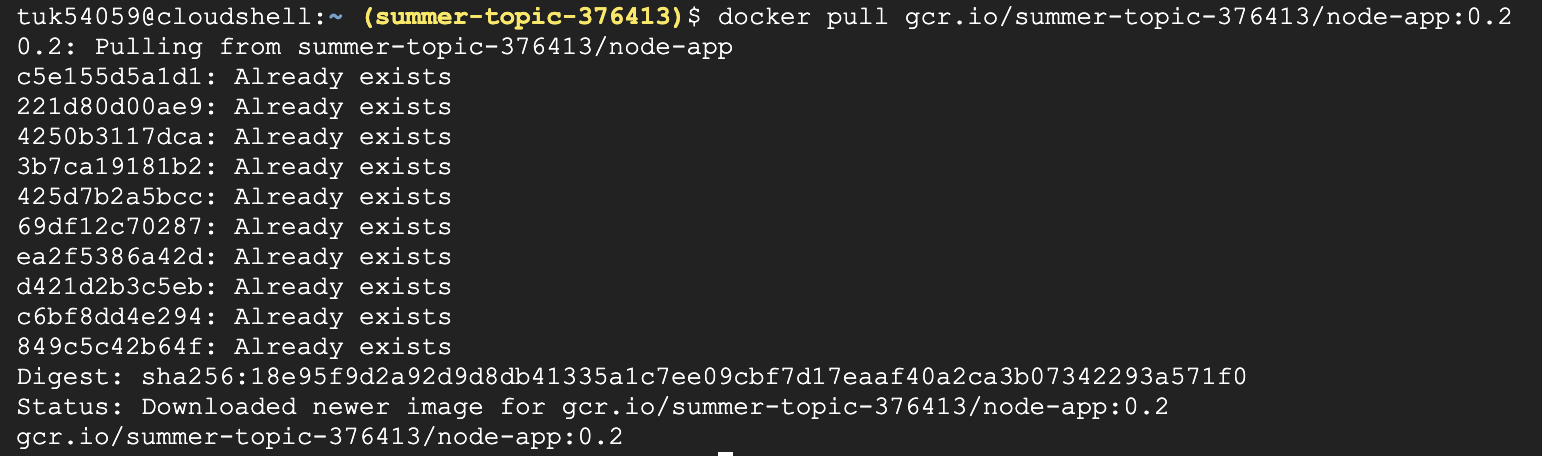
Next, I realize I accidentally did not delete my local image tagged with the gcr prefix, so I deleted it:



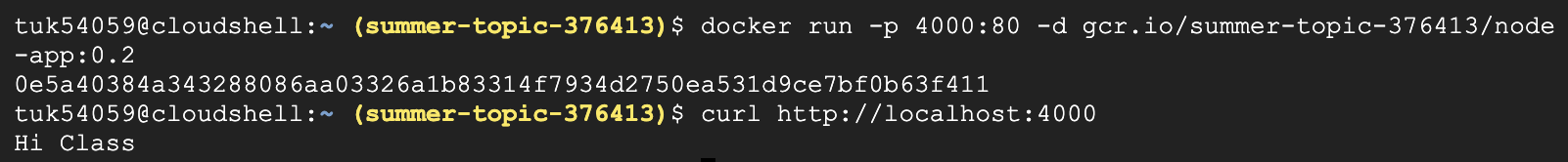
After running the commands above, I no longer having any images locally hosted on my cloud machine:



Next, I will pull the image located in the Google Cloud Container Registry from earlier:



I then ran the registry pulled container:



I was able to see the correct output from the container pulled from the gcr registry. One thing that was confusing to me was that when I pulled the image, it said that certain elements already exist. This is confusing to me because I deleted all of the images stored locally and made sure using the docker images command. I was thinking it might have to do with caching, but I am not sure.