Ben Scuron
Professor Williams
CIS 4360
15 February 2023

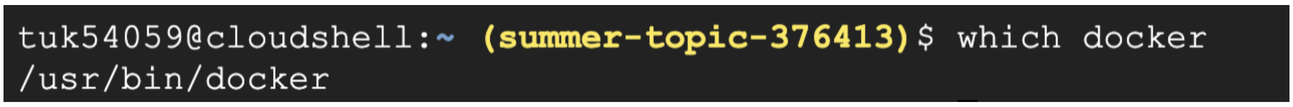
Lab 2 - Containers

Table of Contents

1. Lab 2 - Containers

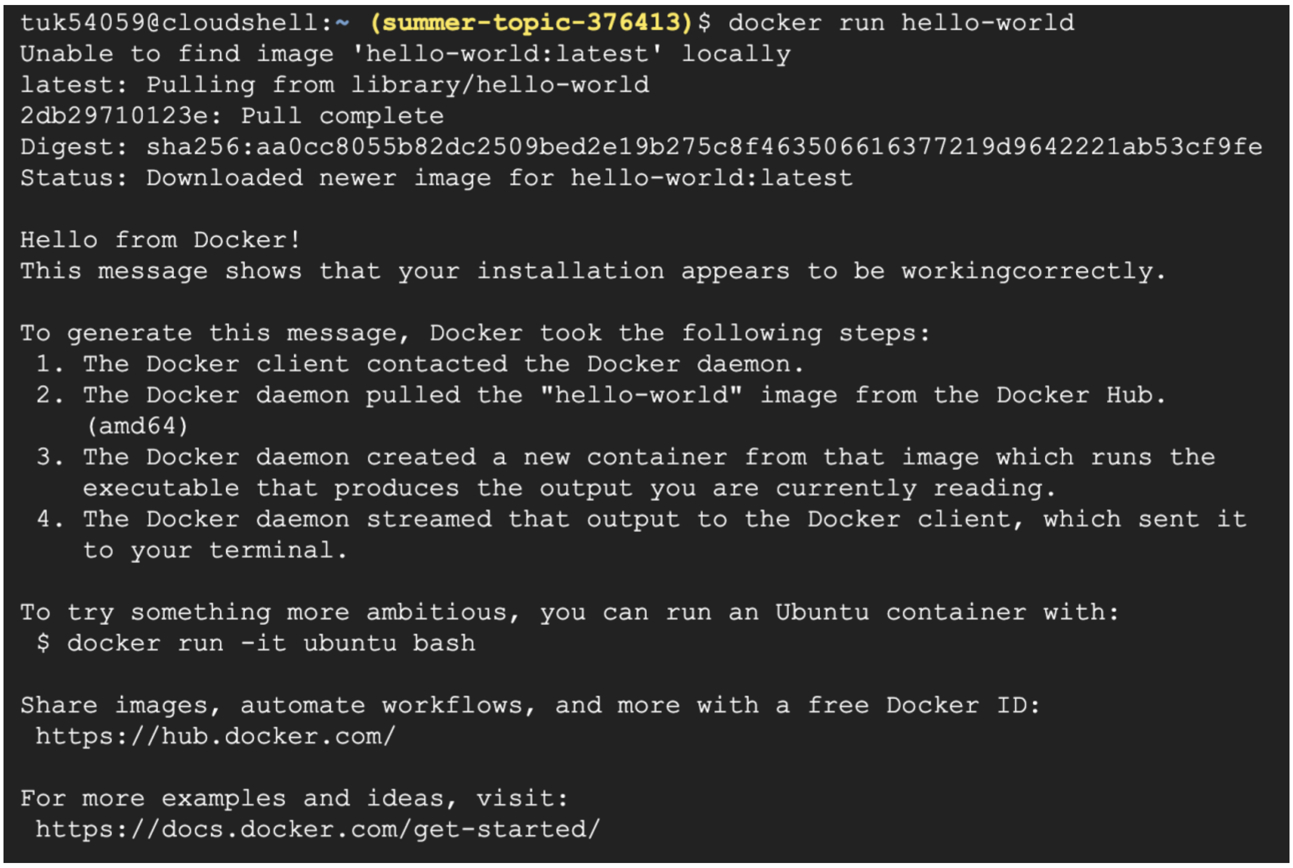
1.1. Hello

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



```
tuk54059@cloudshell:~ (summer-topic-376413)$ which docker
/usr/bin/docker
```

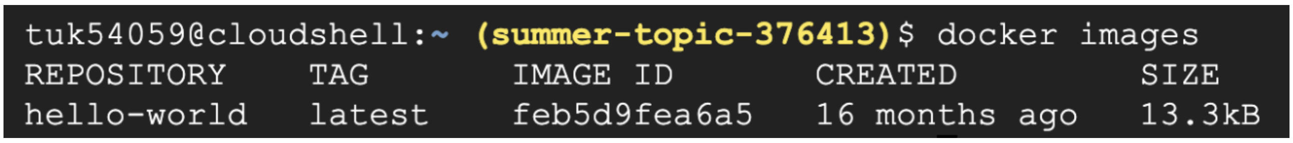
Next, I ran the command to run a hello world Docker container:



```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:aa0cc8055b82dc2509bed2e19b275c8f463506616377219d9642221ab53cf9fe
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be workingcorrectly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

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

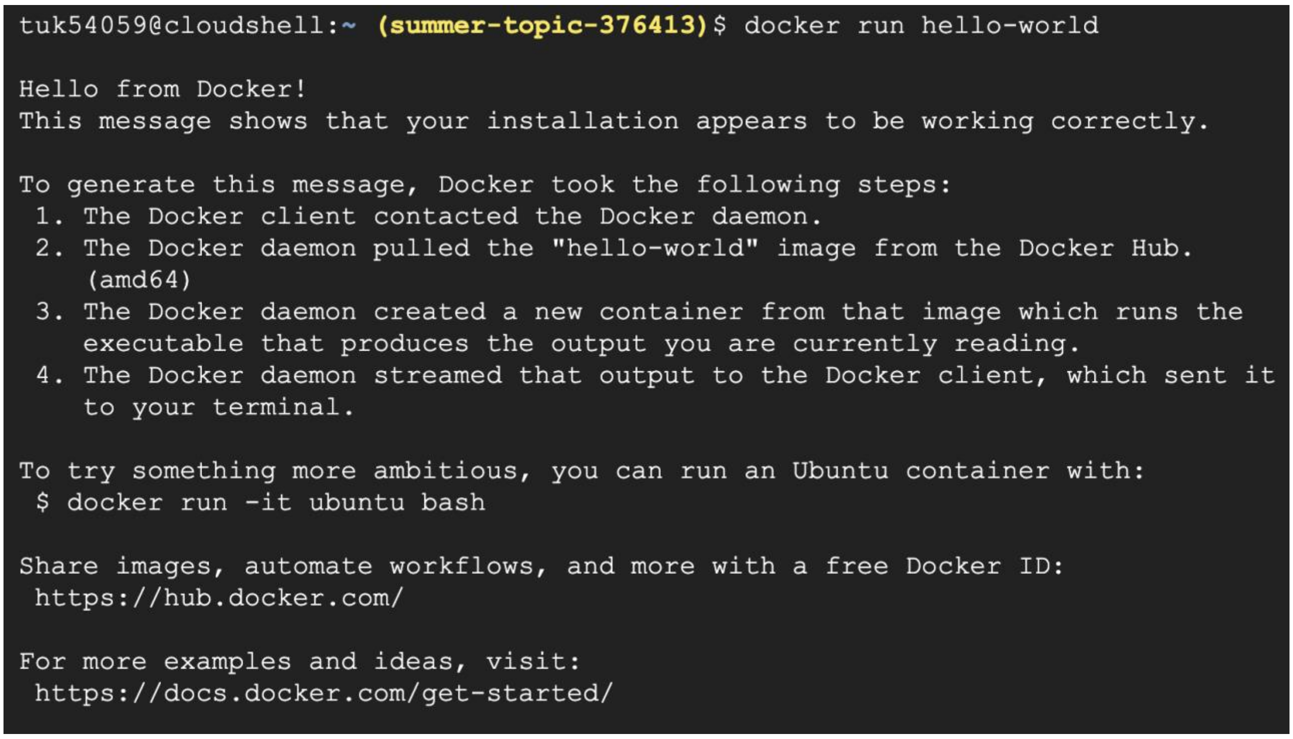
I then ran a command that lists my current Docker images:



```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker images
REPOSITORY     TAG       IMAGE ID       CREATED          SIZE
hello-world    latest    feb5d9fea6a5   16 months ago    13.3kB
```

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.

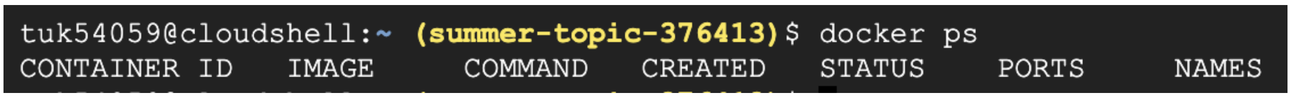
Now, I will try to run the same hello world container. This time, it should not need to pull from the Docker Hub, since the container is stored locally:



```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

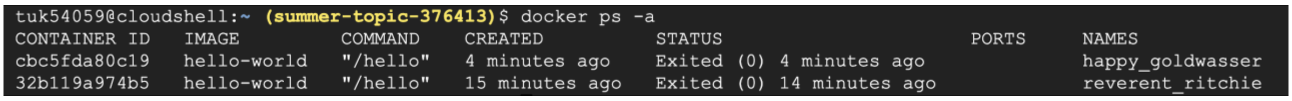
As you can see, Docker was able to find the image locally, without having to pull it down remotely from the Docker Hub registry.

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



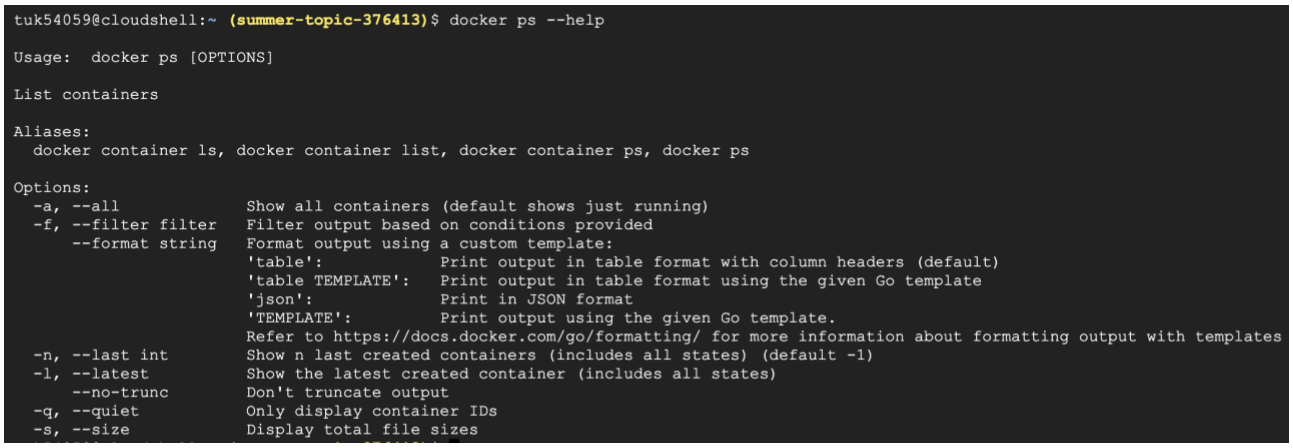
```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

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:



```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker ps -a
CONTAINER ID   IMAGE         COMMAND    CREATED          STATUS                      PORTS     NAMES
cbc5fda80c19   hello-world   "/hello"   4 minutes ago    Exited (0) 4 minutes ago              happy_goldwasser
32b119a974b5   hello-world   "/hello"   15 minutes ago   Exited (0) 14 minutes ago             reverent_ritchie
```

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":

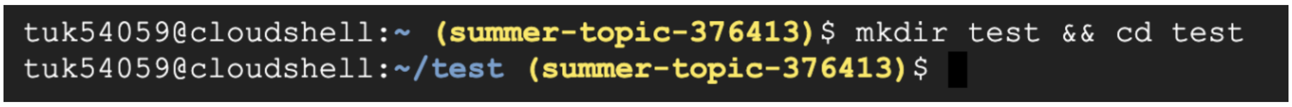


```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker ps --help

Usage:  docker ps [OPTIONS]

List containers

Aliases:
  docker container ls, docker container list, docker container ps, docker ps

Options:
  -a, --all             Show all containers (default shows just running)
  -f, --filter filter   Filter output based on conditions provided
      --format string   Format output using a custom template:
                        'table':            Print output in table format with column headers (default)
                        'table TEMPLATE':   Print output in table format using the given Go template
                        'json':             Print in JSON format
                        'TEMPLATE':         Print output using the given Go template.
                        Refer to https://docs.docker.com/go/formatting/ for more information about formatting output with templates
  -n, --last int        Show n last created containers (includes all states) (default -1)
  -l, --latest          Show the latest created container (includes all states)
      --no-trunc        Don't truncate output
  -q, --quiet           Only display container IDs
  -s, --size            Display total file sizes
```

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)."

1.2. Build

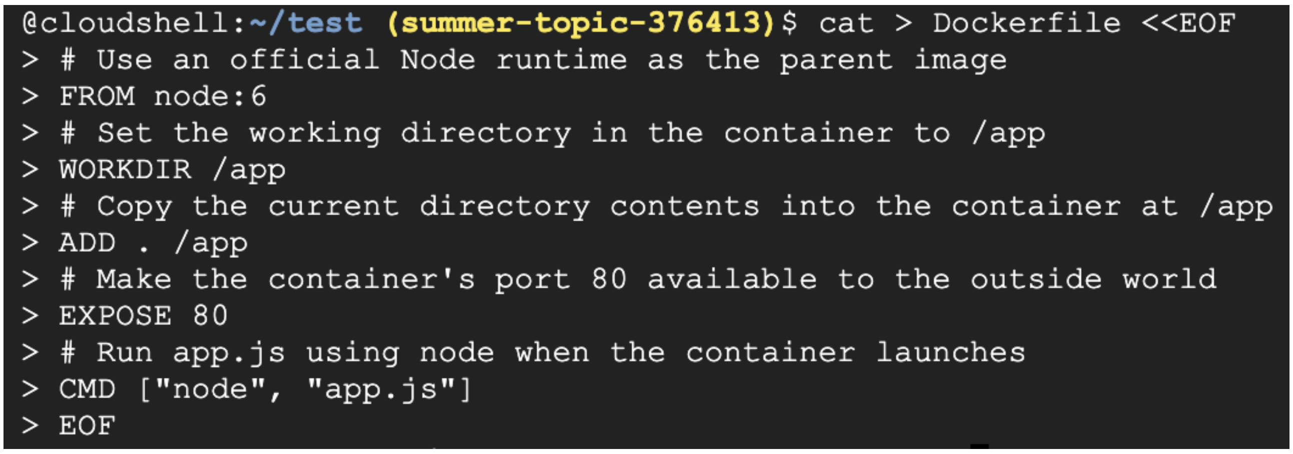
In the build section, we are going to create a Docker image. The first thing we will do is create a directory and change our current directory to that newly created directory:



```
tuk54059@cloudshell:~ (summer-topic-376413)$ mkdir test && cd test
tuk54059@cloudshell:~/test (summer-topic-376413)$ 
```

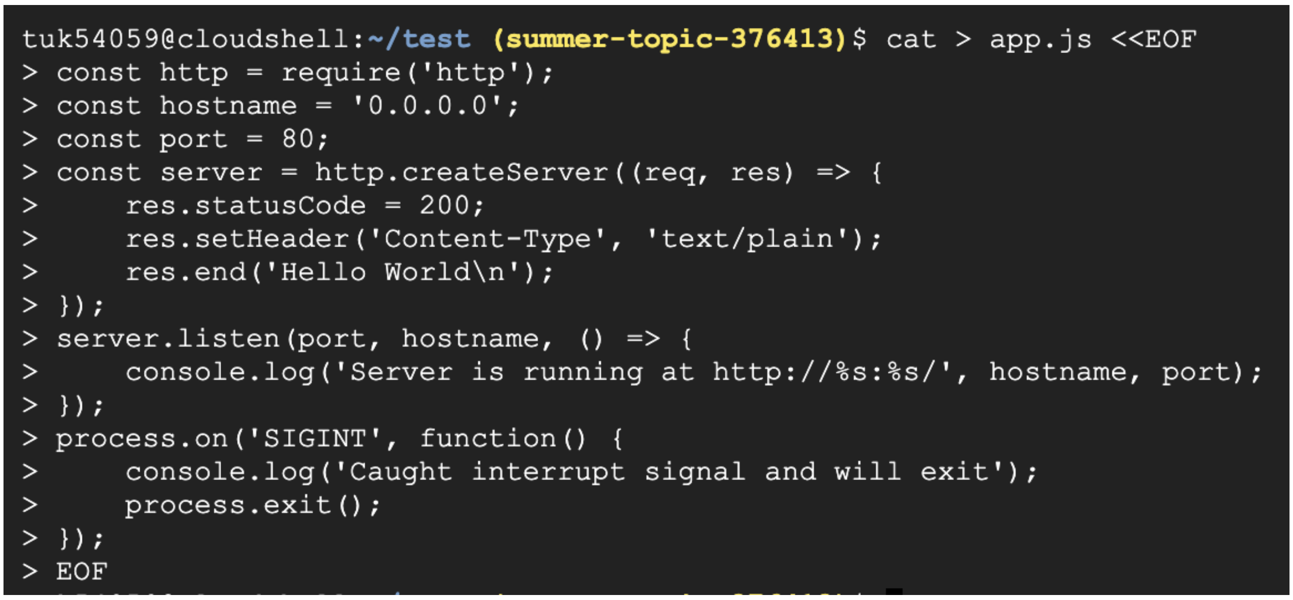
The two ampersands between the mkdir command and cd command is a way to run the second command only if the first command succeeded.

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:



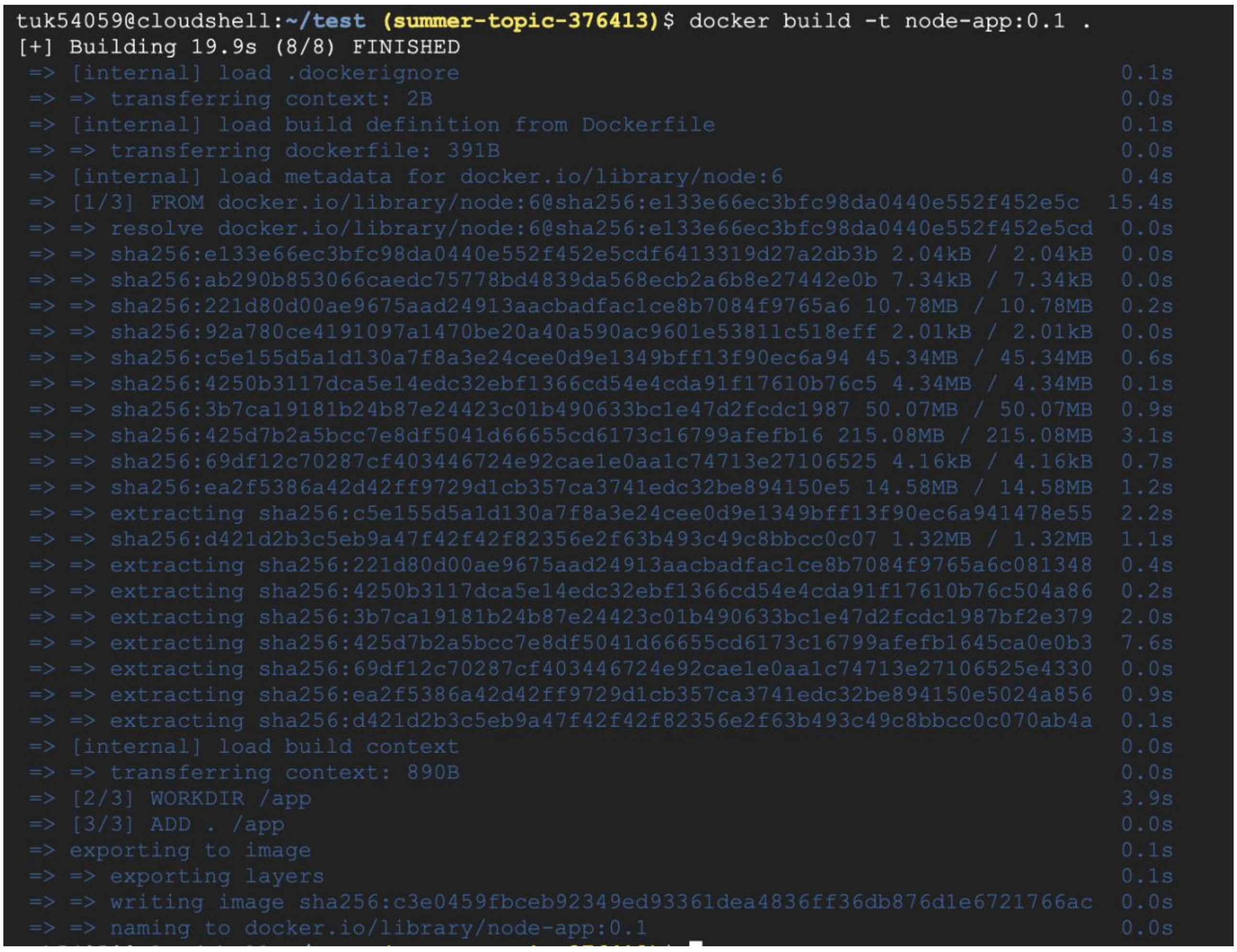
```
@cloudshell:~/test (summer-topic-376413)$ cat > Dockerfile <<EOF
> # Use an official Node runtime as the parent image
> FROM node:6
> # Set the working directory in the container to /app
> WORKDIR /app
> # Copy the current directory contents into the container at /app
> ADD . /app
> # Make the container's port 80 available to the outside world
> EXPOSE 80
> # Run app.js using node when the container launches
> CMD ["node", "app.js"]
> EOF
```

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:



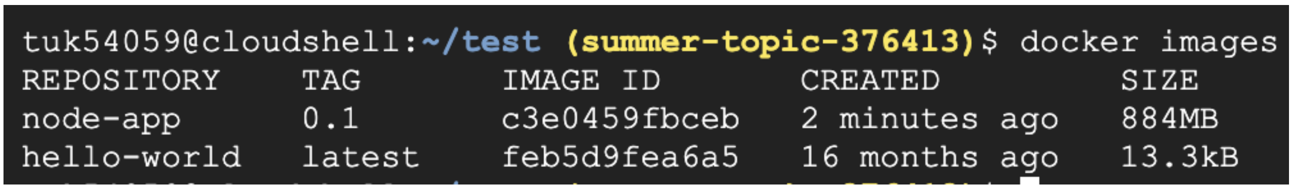
```
tuk54059@cloudshell:~/test (summer-topic-376413)$ cat > app.js <<EOF
> const http = require('http');
> const hostname = '0.0.0.0';
> const port = 80;
> const server = http.createServer((req, res) => {
>     res.statusCode = 200;
>     res.setHeader('Content-Type', 'text/plain');
>     res.end('Hello World\n');
> });
> server.listen(port, hostname, () => {
>     console.log('Server is running at http://%s:%s/', hostname, port);
> });
> process.on('SIGINT', function() {
>     console.log('Caught interrupt signal and will exit');
>     process.exit();
> });
> EOF
```

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



```
tuk54059@cloudshell:~/test (summer-topic-376413)$ docker build -t node-app:0.1 .
[+] Building 19.9s (8/8) FINISHED
 => [internal] load .dockerignore                                                   0.1s
 => => transferring context: 2B                                                     0.0s
 => [internal] load build definition from Dockerfile                                0.1s
 => => transferring dockerfile: 391B                                                0.0s
 => [internal] load metadata for docker.io/library/node:6                           0.4s
 => [1/3] FROM docker.io/library/node:6@sha256:e133e66ec3bfc98da0440e552f452e5c  15.4s
 => => resolve docker.io/library/node:6@sha256:e133e66ec3bfc98da0440e552f452e5cd  0.0s
 => => sha256:e133e66ec3bfc98da0440e552f452e5cdf6413319d27a2db3b 2.04kB / 2.04kB  0.0s
 => => sha256:ab290b853066caedc75778bd4839da568ecb2a6b8e27442e0b 7.34kB / 7.34kB  0.0s
 => => sha256:221d80d00ae9675aad24913aacbadfac1ce8b7084f9765a6 10.78MB / 10.78MB  0.2s
 => => sha256:92a780ce4191097a1470be20a40a590ac9601e53811c518eff 2.01kB / 2.01kB  0.0s
 => => sha256:c5e155d5a1d130a7f8a3e24cee0d9e1349bff13f90ec6a94 45.34MB / 45.34MB  0.6s
 => => sha256:4250b3117dca5e14edc32ebf1366cd54e4cda91f17610b76c5 4.34MB / 4.34MB  0.1s
 => => sha256:3b7ca19181b24b87e24423c01b490633bc1e47d2fcdc1987 50.07MB / 50.07MB  0.9s
 => => sha256:425d7b2a5bcc7e8df5041d66655cd6173c16799afefb16 215.08MB / 215.08MB  3.1s
 => => sha256:69df12c70287cf403446724e92cae1e0aa1c74713e27106525 4.16kB / 4.16kB  0.7s
 => => sha256:ea2f5386a42d42ff9729d1cb357ca3741edc32be894150e5 14.58MB / 14.58MB  1.2s
 => => extracting sha256:c5e155d5a1d130a7f8a3e24cee0d9e1349bff13f90ec6a941478e55  2.2s
 => => sha256:d421d2b3c5eb9a47f42f42f82356e2f63b493c49c8bbcc0c07 1.32MB / 1.32MB  1.1s
 => => extracting sha256:221d80d00ae9675aad24913aacbadfac1ce8b7084f9765a6c081348  0.4s
 => => extracting sha256:4250b3117dca5e14edc32ebf1366cd54e4cda91f17610b76c504a86  0.2s
 => => extracting sha256:3b7ca19181b24b87e24423c01b490633bc1e47d2fcdc1987bf2e379  2.0s
 => => extracting sha256:425d7b2a5bcc7e8df5041d66655cd6173c16799afefb1645ca0e0b3  7.6s
 => => extracting sha256:69df12c70287cf403446724e92cae1e0aa1c74713e27106525e4330  0.0s
 => => extracting sha256:ea2f5386a42d42ff9729d1cb357ca3741edc32be894150e5024a856  0.9s
 => => extracting sha256:d421d2b3c5eb9a47f42f42f82356e2f63b493c49c8bbcc0c070ab4a  0.1s
 => [internal] load build context                                                   0.0s
 => => transferring context: 890B                                                   0.0s
 => [2/3] WORKDIR /app                                                              3.9s
 => [3/3] ADD . /app                                                                0.0s
 => exporting to image                                                              0.1s
 => => exporting layers                                                             0.1s
 => => writing image sha256:c3e0459fbceb92349ed93361dea4836ff36db876d1e6721766ac  0.0s
 => => naming to docker.io/library/node-app:0.1                                     0.0s
```

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

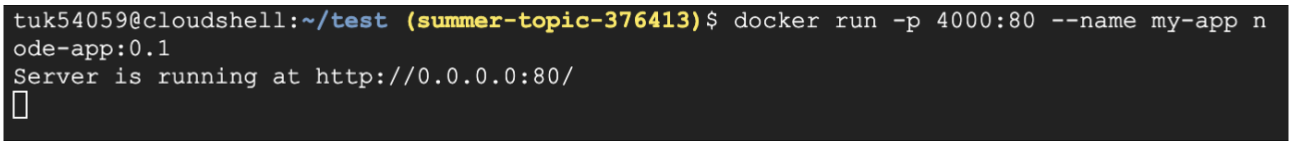


```
tuk54059@cloudshell:~/test (summer-topic-376413)$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
node-app      0.1       c3e0459fbceb   2 minutes ago    884MB
hello-world   latest    feb5d9fea6a5   16 months ago    13.3kB
```

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

1.3. *Run*

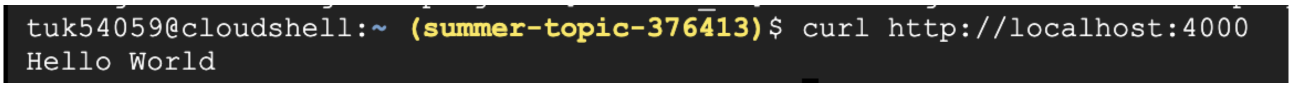
Now that we have successfully built the Docker image, it is time to run it. Remember, the web server will run on port 80.



```
tuk54059@cloudshell:~/test (summer-topic-376413)$ docker run -p 4000:80 --name my-app n
ode-app:0.1
Server is running at http://0.0.0.0:80/
```

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

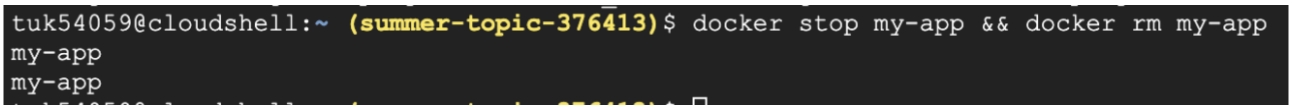
Now to test if I can connect to the web server, I ran the curl command in another console instance:



```
tuk54059@cloudshell:~ (summer-topic-376413)$ curl http://localhost:4000
Hello World
```

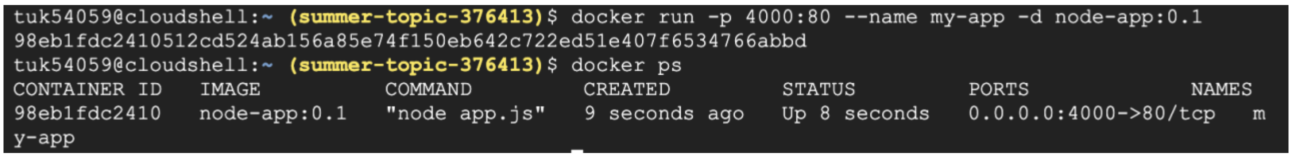
After connecting to our web server that we created, I received the Hello World message that we were expecting.

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



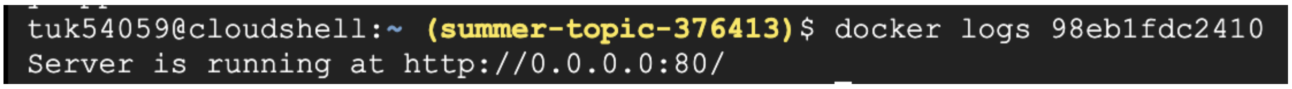
```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker stop my-app && docker rm my-app
my-app
my-app
```

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:



```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker run -p 4000:80 --name my-app -d node-app:0.1
98eb1fdc2410512cd524ab156a85e74f150eb642c722ed51e407f6534766abbd
tuk54059@cloudshell:~ (summer-topic-376413)$ docker ps
CONTAINER ID   IMAGE          COMMAND         CREATED         STATUS         PORTS                  NAMES
98eb1fdc2410   node-app:0.1   "node app.js"   9 seconds ago   Up 8 seconds   0.0.0.0:4000->80/tcp   m
y-app
```

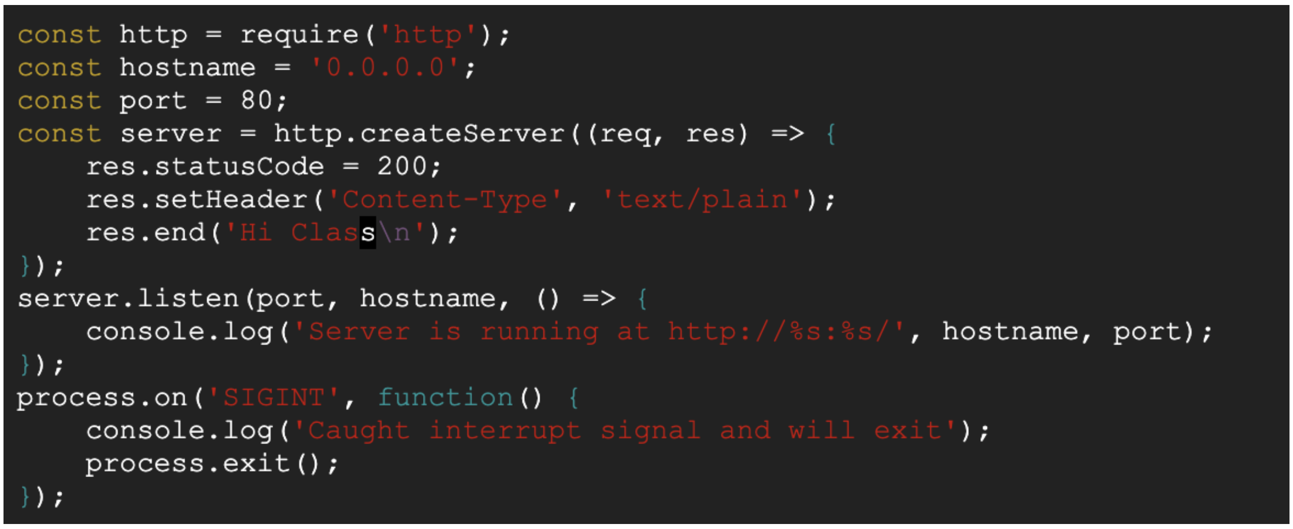
To check the output of a container running in the background, we can use the docker logs command. To see the logs of a specific container, we pass the container id to the docker logs command:



```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker logs 98eb1fdc2410
Server is running at http://0.0.0.0:80/
```

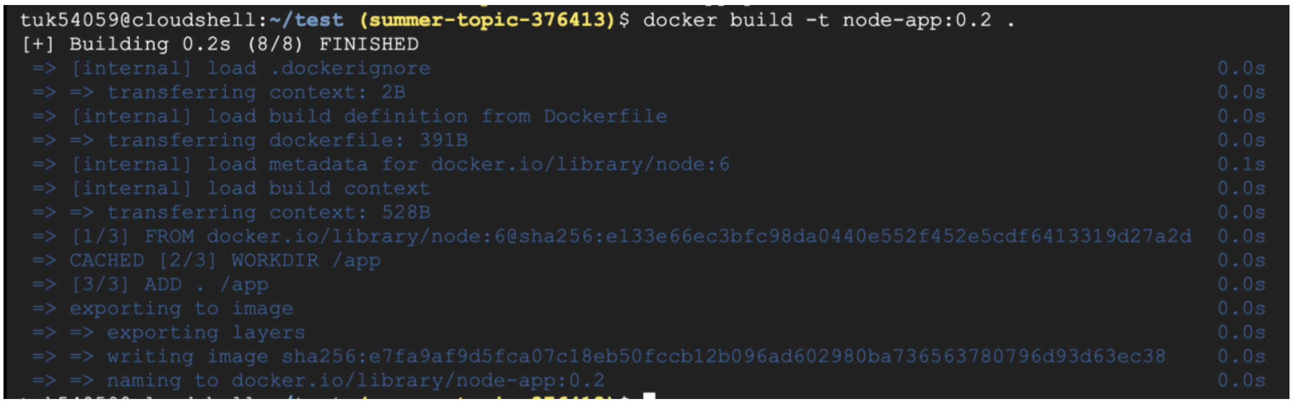
As you can see from the output of Docker logs, the container running our web server is running on port 80 just as expected.

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:



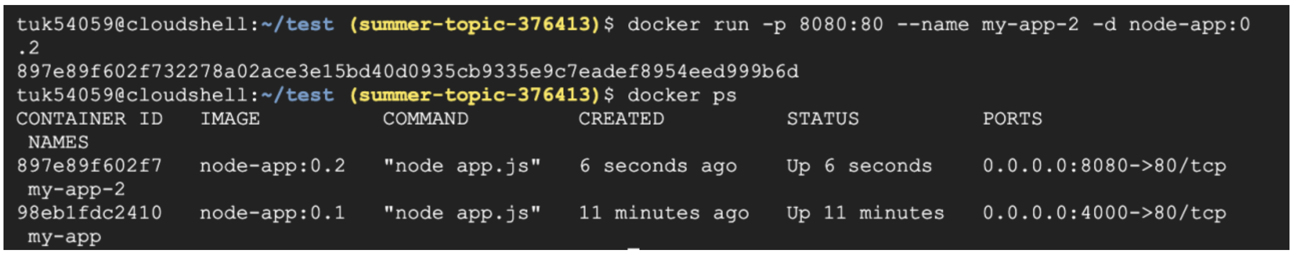
```
const http = require('http');
const hostname = '0.0.0.0';
const port = 80;
const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hi Class\n');
});
server.listen(port, hostname, () => {
    console.log('Server is running at http://%s:%s/', hostname, port);
});
process.on('SIGINT', function() {
    console.log('Caught interrupt signal and will exit');
    process.exit();
});
```

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



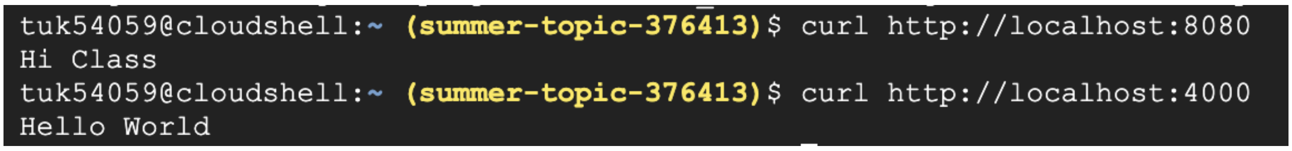
```
tuk54059@cloudshell:~/test (summer-topic-376413)$ docker build -t node-app:0.2 .
[+] Building 0.2s (8/8) FINISHED
 => [internal] load .dockerignore                                                    0.0s
 => => transferring context: 2B                                                      0.0s
 => [internal] load build definition from Dockerfile                                 0.0s
 => => transferring dockerfile: 391B                                                 0.0s
 => [internal] load metadata for docker.io/library/node:6                            0.1s
 => [internal] load build context                                                    0.0s
 => => transferring context: 528B                                                    0.0s
 => [1/3] FROM docker.io/library/node:6@sha256:e133e66ec3bfc98da0440e552f452e5cdf6413319d27a2d  0.0s
 => CACHED [2/3] WORKDIR /app                                                        0.0s
 => [3/3] ADD . /app                                                                 0.0s
 => exporting to image                                                               0.0s
 => => exporting layers                                                              0.0s
 => => writing image sha256:e7fa9af9d5fca07c18eb50fccb12b096ad602980ba736563780796d93d63ec38  0.0s
 => => naming to docker.io/library/node-app:0.2                                      0.0s
```

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



```
tuk54059@cloudshell:~/test (summer-topic-376413)$ docker run -p 8080:80 --name my-app-2 -d node-app:0
.2
897e89f602f732278a02ace3e15bd40d0935cb9335e9c7eadef8954eed999b6d
tuk54059@cloudshell:~/test (summer-topic-376413)$ docker ps
CONTAINER ID   IMAGE          COMMAND         CREATED          STATUS          PORTS
 NAMES
897e89f602f7   node-app:0.2   "node app.js"   6 seconds ago    Up 6 seconds    0.0.0.0:8080->80/tcp
 my-app-2
98eb1fdc2410   node-app:0.1   "node app.js"   11 minutes ago   Up 11 minutes   0.0.0.0:4000->80/tcp
 my-app
```

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:

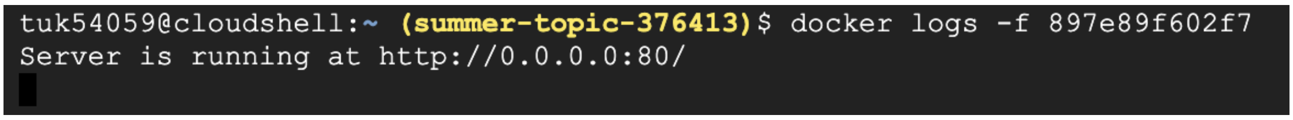


```
tuk54059@cloudshell:~ (summer-topic-376413)$ curl http://localhost:8080
Hi Class
tuk54059@cloudshell:~ (summer-topic-376413)$ curl http://localhost:4000
Hello World
```

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.

1.4. Debug

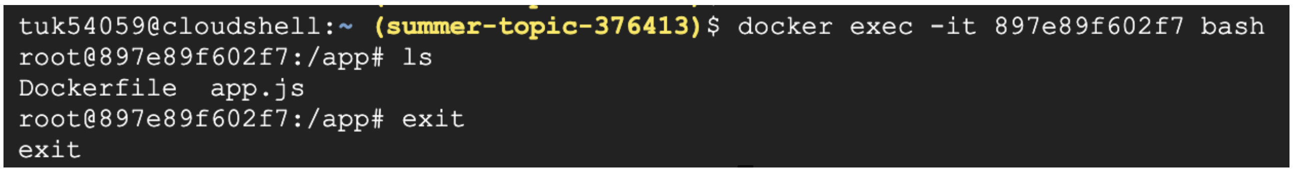
In this section, I try to familiarize myself with debugging Docker containers. One useful command is the docker logs command that we used previously:



```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker logs -f 897e89f602f7
Server is running at http://0.0.0.0:80/
```

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

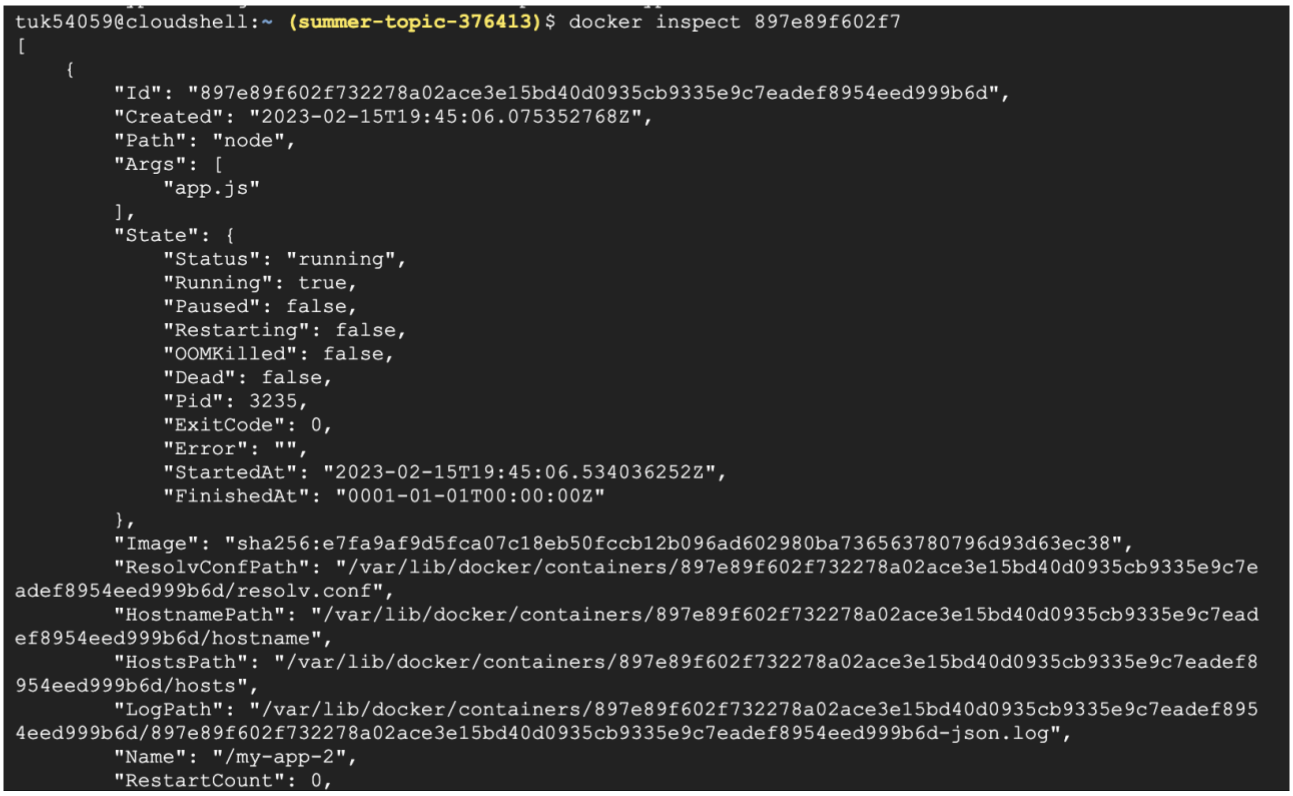
Sometimes it may be useful to create an interactive Bash shell for a Docker container. To do this, I ran the following command:



```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker exec -it 897e89f602f7 bash
root@897e89f602f7:/app# ls
Dockerfile  app.js
root@897e89f602f7:/app# exit
exit
```

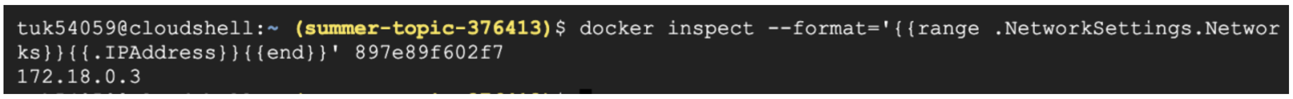
This is useful for when you need to run a series of commands in a Docker container. This interactive shell allows you to easily navigate within the Docker container itself.

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



```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker inspect 897e89f602f7
[
    {
        "Id": "897e89f602f732278a02ace3e15bd40d0935cb9335e9c7eadef8954eed999b6d",
        "Created": "2023-02-15T19:45:06.075352768Z",
        "Path": "node",
        "Args": [
            "app.js"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 3235,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2023-02-15T19:45:06.534036252Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image": "sha256:e7fa9af9d5fca07c18eb50fccb12b096ad602980ba736563780796d93d63ec38",
        "ResolvConfPath": "/var/lib/docker/containers/897e89f602f732278a02ace3e15bd40d0935cb9335e9c7e
adef8954eed999b6d/resolv.conf",
        "HostnamePath": "/var/lib/docker/containers/897e89f602f732278a02ace3e15bd40d0935cb9335e9c7ead
ef8954eed999b6d/hostname",
        "HostsPath": "/var/lib/docker/containers/897e89f602f732278a02ace3e15bd40d0935cb9335e9c7eadef8
954eed999b6d/hosts",
        "LogPath": "/var/lib/docker/containers/897e89f602f732278a02ace3e15bd40d0935cb9335e9c7eadef895
4eed999b6d/897e89f602f732278a02ace3e15bd40d0935cb9335e9c7eadef8954eed999b6d-json.log",
        "Name": "/my-app-2",
        "RestartCount": 0,
```

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:

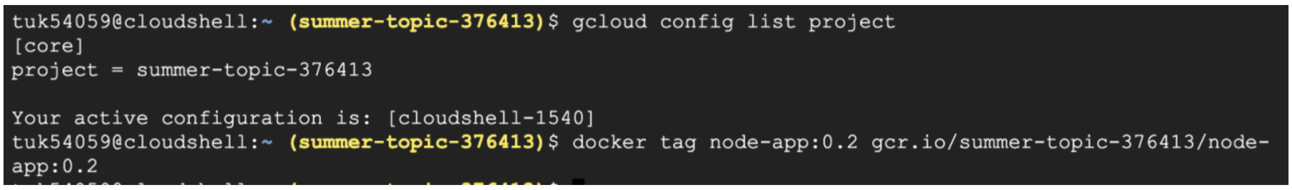


```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker inspect --format='{{range .NetworkSettings.Networ
ks}}{{.IPAddress}}{{end}}' 897e89f602f7
172.18.0.3
```

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

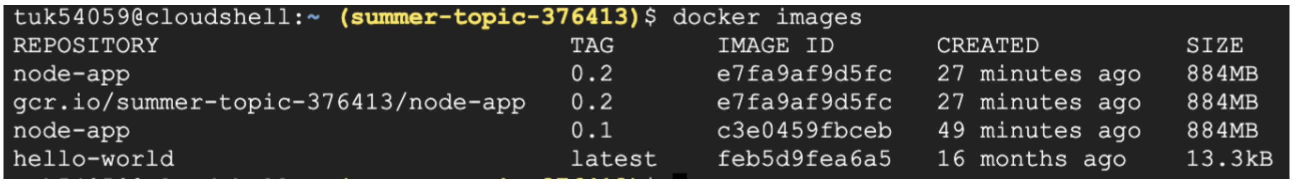
1.5. Publish

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:



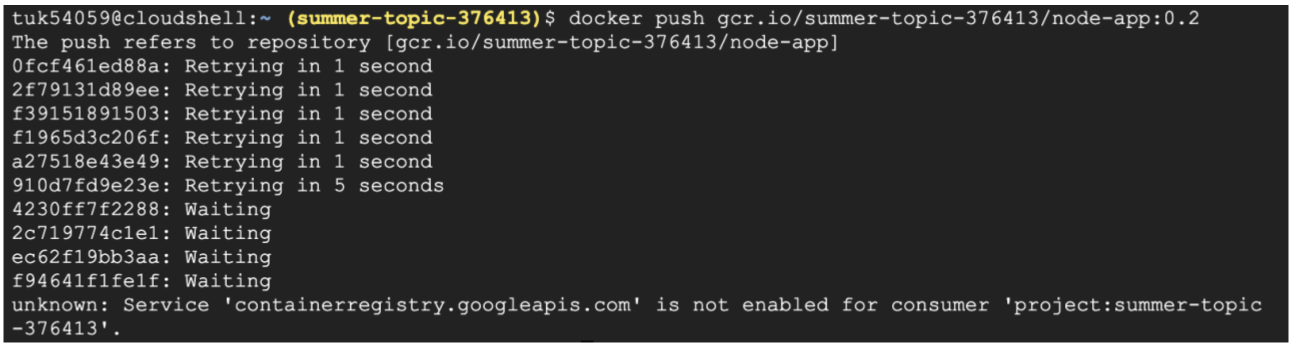
```
tuk54059@cloudshell:~ (summer-topic-376413)$ gcloud config list project
[core]
project = summer-topic-376413

Your active configuration is: [cloudshell-1540]
tuk54059@cloudshell:~ (summer-topic-376413)$ docker tag node-app:0.2 gcr.io/summer-topic-376413/node-
app:0.2
```

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



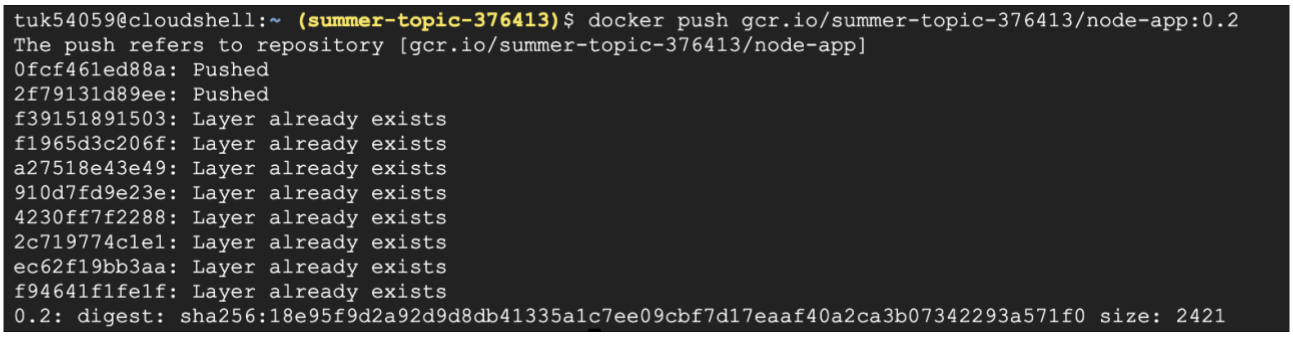
```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker images
REPOSITORY                            TAG       IMAGE ID       CREATED          SIZE
node-app                              0.2       e7fa9af9d5fc   27 minutes ago   884MB
gcr.io/summer-topic-376413/node-app   0.2       e7fa9af9d5fc   27 minutes ago   884MB
node-app                              0.1       c3e0459fbceb   49 minutes ago   884MB
hello-world                           latest    feb5d9fea6a5   16 months ago    13.3kB
```

Next, I finally tried to publish the image:



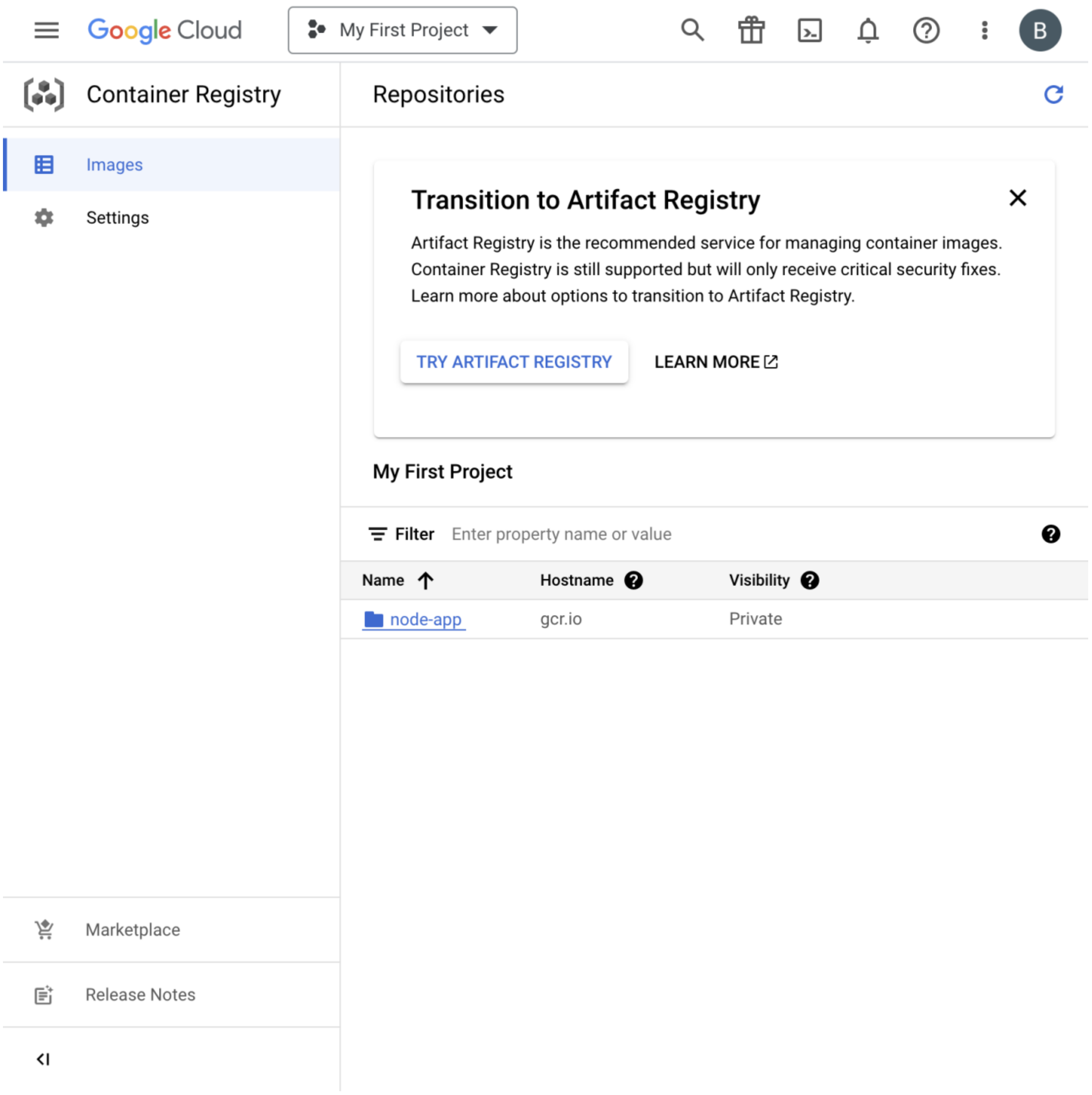
```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker push gcr.io/summer-topic-376413/node-app:0.2
The push refers to repository [gcr.io/summer-topic-376413/node-app]
0fcf461ed88a: Retrying in 1 second
2f79131d89ee: Retrying in 1 second
f39151891503: Retrying in 1 second
f1965d3c206f: Retrying in 1 second
a27518e43e49: Retrying in 1 second
910d7fd9e23e: Retrying in 5 seconds
4230ff7f2288: Waiting
2c719774c1e1: Waiting
ec62f19bb3aa: Waiting
f94641f1fe1f: Waiting
unknown: Service 'containerregistry.googleapis.com' is not enabled for consumer 'project:summer-topic
-376413'.
```

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:



```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker push gcr.io/summer-topic-376413/node-app:0.2
The push refers to repository [gcr.io/summer-topic-376413/node-app]
0fcf461ed88a: Pushed
2f79131d89ee: Pushed
f39151891503: Layer already exists
f1965d3c206f: Layer already exists
a27518e43e49: Layer already exists
910d7fd9e23e: Layer already exists
4230ff7f2288: Layer already exists
2c719774c1e1: Layer already exists
ec62f19bb3aa: Layer already exists
f94641f1fe1f: Layer already exists
0.2: digest: sha256:18e95f9d2a92d9d8db41335a1c7ee09cbf7d17eaaf40a2ca3b07342293a571f0 size: 2421
```

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



Repositories

Transition to Artifact Registry

Artifact Registry is the recommended service for managing container images. Container Registry is still supported but will only receive critical security fixes. Learn more about options to transition to Artifact Registry.

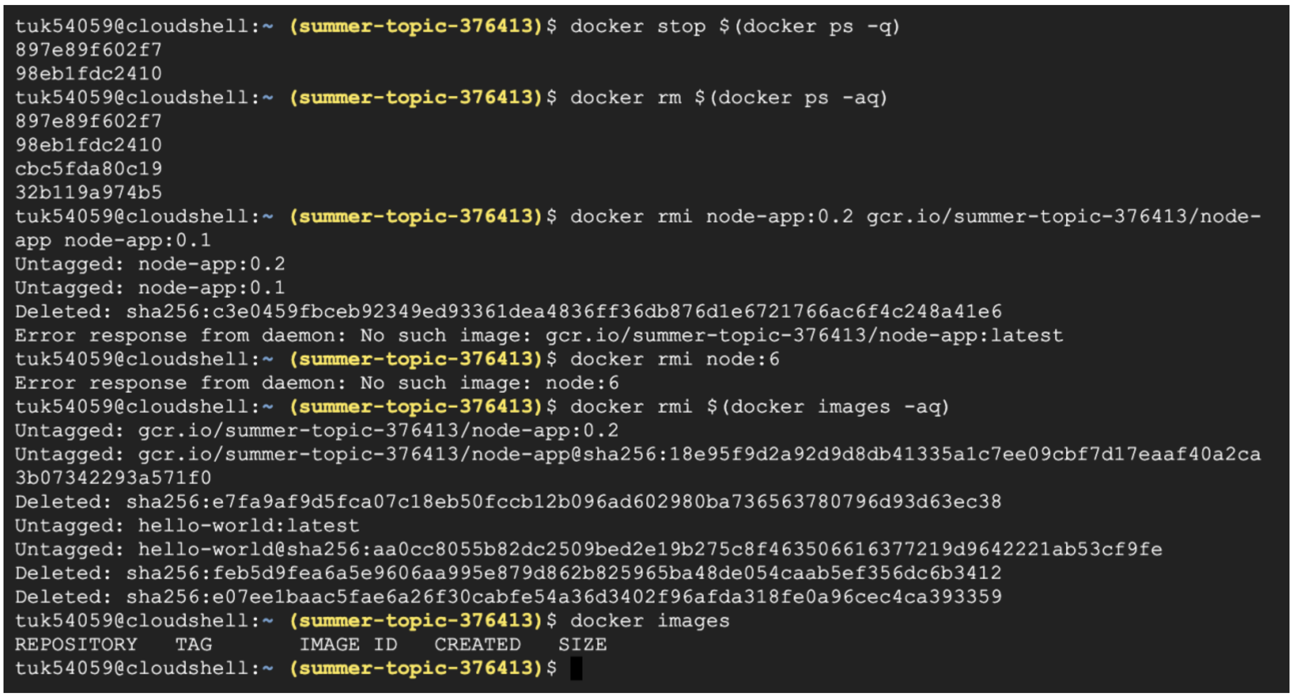
TRY ARTIFACT REGISTRY LEARN MORE

My First Project

Filter Enter property name or value

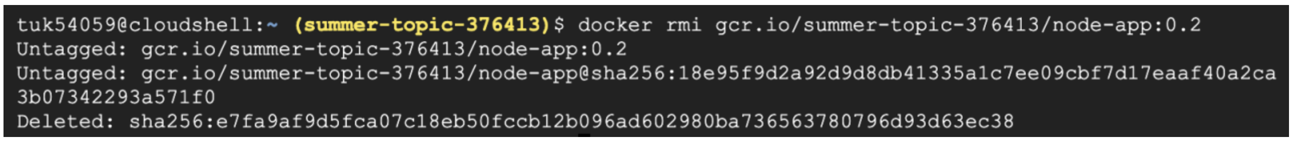
Name ↑	Hostname	Visibility
node-app	gcr.io	Private

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



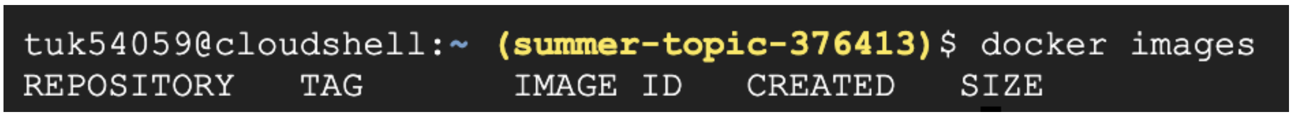
```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker stop $(docker ps -q)
897e89f602f7
98eb1fdc2410
tuk54059@cloudshell:~ (summer-topic-376413)$ docker rm $(docker ps -aq)
897e89f602f7
98eb1fdc2410
cbc5fda80c19
32b119a974b5
tuk54059@cloudshell:~ (summer-topic-376413)$ docker rmi node-app:0.2 gcr.io/summer-topic-376413/node-
app node-app:0.1
Untagged: node-app:0.2
Untagged: node-app:0.1
Deleted: sha256:c3e0459fbceb92349ed93361dea4836ff36db876d1e6721766ac6f4c248a41e6
Error response from daemon: No such image: gcr.io/summer-topic-376413/node-app:latest
tuk54059@cloudshell:~ (summer-topic-376413)$ docker rmi node:6
Error response from daemon: No such image: node:6
tuk54059@cloudshell:~ (summer-topic-376413)$ docker rmi $(docker images -aq)
Untagged: gcr.io/summer-topic-376413/node-app:0.2
Untagged: gcr.io/summer-topic-376413/node-app@sha256:18e95f9d2a92d9d8db41335a1c7ee09cbf7d17eaaf40a2ca
3b07342293a571f0
Deleted: sha256:e7fa9af9d5fca07c18eb50fccb12b096ad602980ba736563780796d93d63ec38
Untagged: hello-world:latest
Untagged: hello-world@sha256:aa0cc8055b82dc2509bed2e19b275c8f463506616377219d9642221ab53cf9fe
Deleted: sha256:feb5d9fea6a5e9606aa995e879d862b825965ba48de054caab5ef356dc6b3412
Deleted: sha256:e07ee1baac5fae6a26f30cabfe54a36d3402f96afda318fe0a96cec4ca393359
tuk54059@cloudshell:~ (summer-topic-376413)$ docker images
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
tuk54059@cloudshell:~ (summer-topic-376413)$
```

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



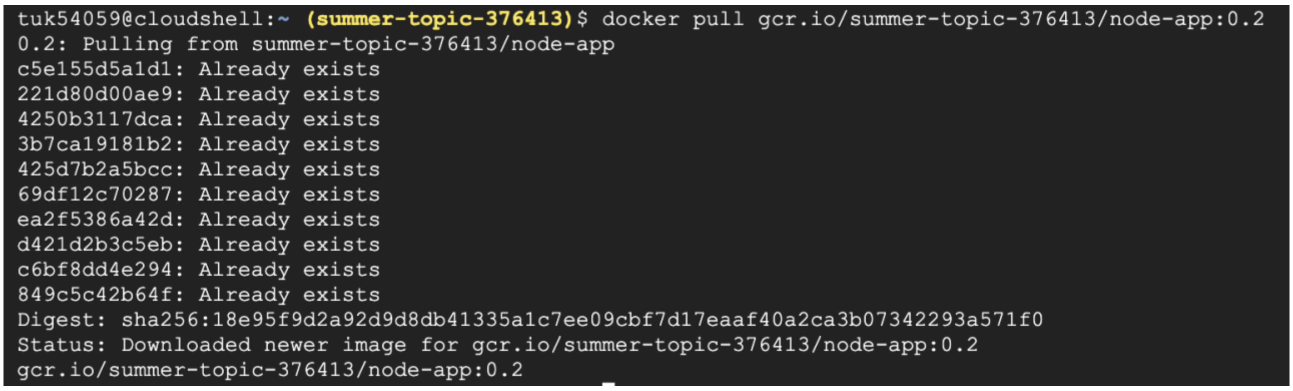
```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker rmi gcr.io/summer-topic-376413/node-app:0.2
Untagged: gcr.io/summer-topic-376413/node-app:0.2
Untagged: gcr.io/summer-topic-376413/node-app@sha256:18e95f9d2a92d9d8db41335a1c7ee09cbf7d17eaaf40a2ca
3b07342293a571f0
Deleted: sha256:e7fa9af9d5fca07c18eb50fccb12b096ad602980ba736563780796d93d63ec38
```

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



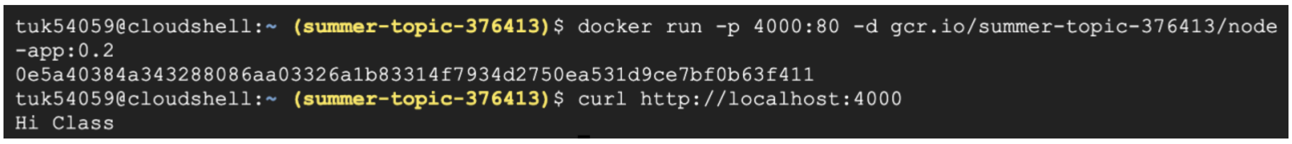
```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker images
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
```

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



```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker pull gcr.io/summer-topic-376413/node-app:0.2
0.2: Pulling from summer-topic-376413/node-app
c5e155d5a1d1: Already exists
221d80d00ae9: Already exists
4250b3117dca: Already exists
3b7ca19181b2: Already exists
425d7b2a5bcc: Already exists
69df12c70287: Already exists
ea2f5386a42d: Already exists
d421d2b3c5eb: Already exists
c6bf8dd4e294: Already exists
849c5c42b64f: Already exists
Digest: sha256:18e95f9d2a92d9d8db41335a1c7ee09cbf7d17eaaf40a2ca3b07342293a571f0
Status: Downloaded newer image for gcr.io/summer-topic-376413/node-app:0.2
gcr.io/summer-topic-376413/node-app:0.2
```

I then ran the registry pulled container:



```
tuk54059@cloudshell:~ (summer-topic-376413)$ docker run -p 4000:80 -d gcr.io/summer-topic-376413/node
-app:0.2
0e5a40384a343288086aa03326a1b83314f7934d2750ea531d9ce7bf0b63f411
tuk54059@cloudshell:~ (summer-topic-376413)$ curl http://localhost:4000
Hi Class
```

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.