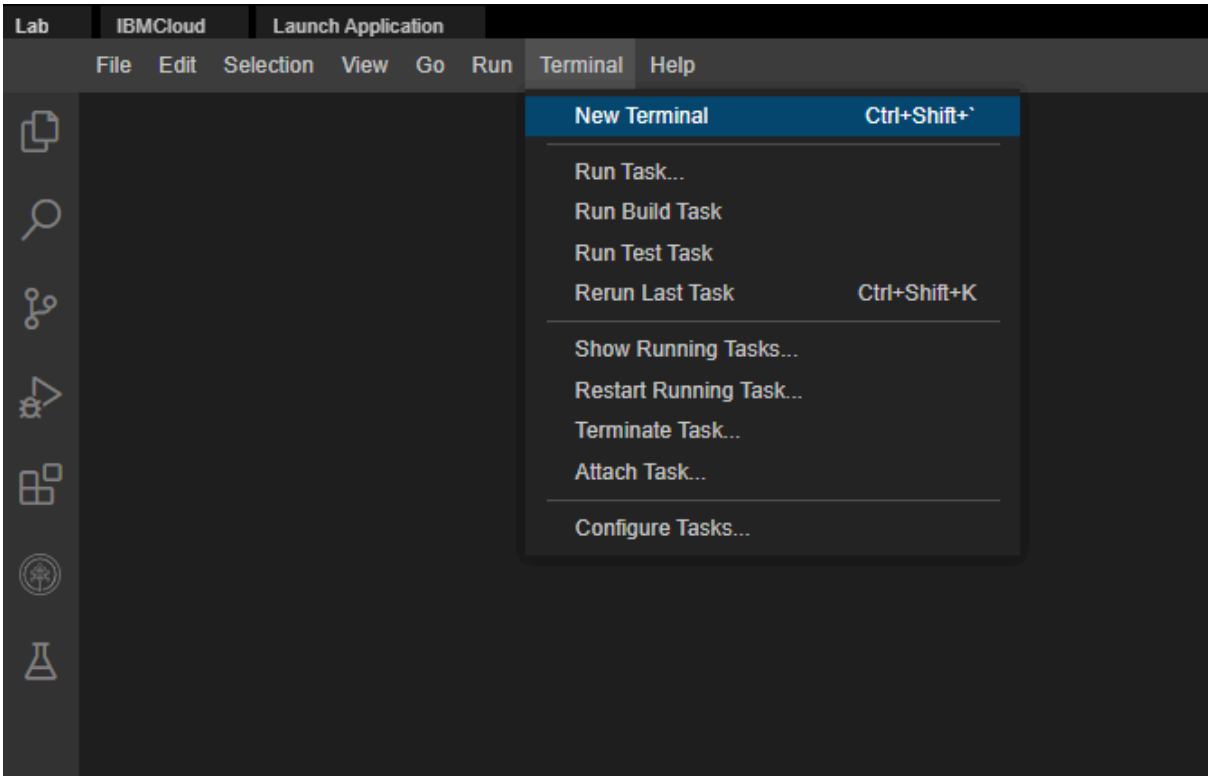
Verify the environment and command line tools

Open a terminal window by using the menu in the editor: Terminal > New Terminal.

Note:If the terminal is already opened, please skip this step.



2. Verify that docker CLI is installed.

1

- docker --version!

You should see the following output, although the version may be different:

```
theia@theiadocker-[redacted]:/home/project$ docker --version
Docker version 20.10.7, build 20.10.7-0ubuntu5~18.04.3
```

3. Verify that ibmcloud CLI is installed.

- ibmcloud version

You should see the following output, although the version may be different:

```
theia@theiadocker-[redacted]:/home/project$ ibmcloud version
ibmcloud version 2.1.1+19d7e02-2021-09-24T15:16:38+00:00
```

4. Change to your project folder.

   Note: If you are already on the '/home/project' folder, please skip this step.

- cd /home/project

5. Clone the git repository that contains the artifacts needed for this lab, if it doesn't already exist.

- [! -d 'CC201'] && git clone https://github.com/ibm-developer-skills-network/CC201.git

```
theia@theiadocker-[redacted]:/home/project$ git clone https://github.com/ibm-developer-skills-network/CC201.git
Cloning into 'CC201'...
remote: Enumerating objects: 20, done.
remote: Counting objects: 100% (20/20), done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 20 (delta 6), reused 19 (delta 6), pack-reused 0
Unpacking objects: 100% (20/20), done.
```

6. Change to the directory for this lab by running the following command. cd will change the working/current directory to the directory with the name specified, in this case **CC201/labs/1_ContainersAndDcoker**.

- cd CC201/labs/1_ContainersAndDocker/

7. List the contents of this directory to see the artifacts for this lab.

- ls

```
theia@theiadocker-[redacted]:/home/project/CC201/labs/1_ContainersAndDocker$ ls
app.js  Dockerfile  package.json
```

Pull an image from Docker Hub and run it as a container

1. Use the docker CLI to list your images.

- docker images

You should see an empty table (with only headings) since you don't have any images yet.

```
theia@theiadocker-[redacted]:/home/project/CC201/labs/1_ContainersAndDocker$ docker images
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
```

2. Pull your first image from Docker Hub.

- docker pull hello-world

```
theia@theiadocker-[redacted]:/home/project/CC201/labs/1_ContainersAndDocker$ docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:bfea6278a0a267fad2634554f4f0c6f31981eea41c553fdf5a83e95a41d40c38
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

3. List images again.

- docker images

You should now see the hello-world image present in the table.

```
theia@theiadocker-[redacted]:/home/project/CC201/labs/1_ContainersAndDocker$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
hello-world   latest    feb5d9fea6a5   6 months ago   13.3kB
```

4. Run the hello-world image as a container.

- docker run hello-world

You should see a **'Hello from Docker!'** message.

```
theia@theiadocker-[redacted]:/home/project/CC201/labs/1_ContainersAndDocker$ 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/
```

There will also be an explanation of what Docker did to generate this message.

5. List the containers to see that your container ran and exited successfully.

- docker ps -a

```
theia@theiadocker-[redacted]:/home/project/CC201/labs/1_ContainersAndDocker$ docker ps -a
CONTAINER ID   IMAGE         COMMAND    CREATED          STATUS                      PORTS     NAMES
5e1756c09910   hello-world   "/hello"   8 seconds ago    Exited (0) 6 seconds ago              trusting_bose
```

Among other things, for this container you should see a container ID, the image name (hello-world), and a status that indicates that the container exited successfully.

6. Note the CONTAINER ID from the previous output and replace the **<container_id>** tag in the command below with this value. This command removes your container.

- docker container rm <container_id>

```
theia@theiadocker-[redacted]:/home/project/CC201/labs/1_ContainersAndDocker$ docker container rm 5e1756c09910
5e1756c09910
```

7. Verify that that the container has been removed. Run the following command.

- docker ps -a

```
theia@theiadocker-[redacted]:/home/project/CC201/labs/1_ContainersAndDocker$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

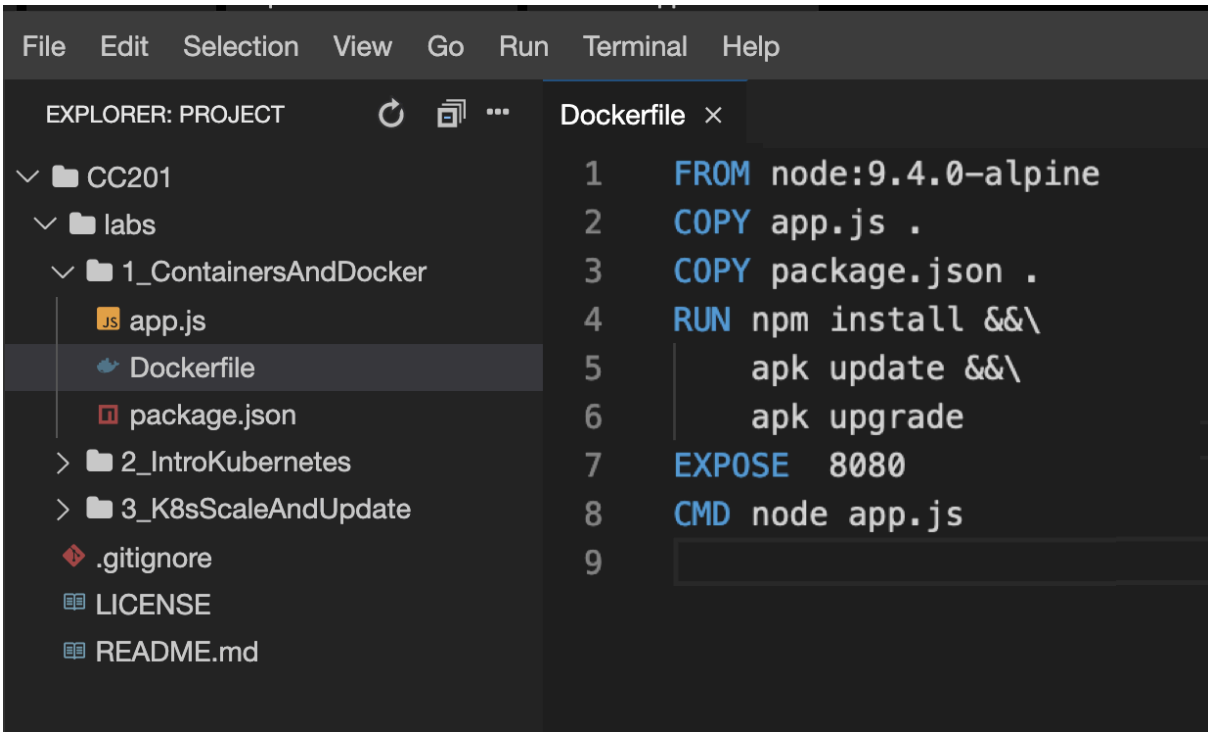
Congratulations on pulling an image from Docker Hub and running your first container! Now let's try and build our own image.

Build an image using a Dockerfile

1. The current working directory contains a simple Node.js application that we will run in a container. The app will print a hello message along with the hostname. The following files are needed to run the app in a container:

- app.js is the main application, which simply replies with a hello world message.
- package.json defines the dependencies of the application.
- Dockerfile defines the instructions Docker uses to build the image.

2. Use the Explorer to view the files needed for this app. Click the Explorer icon (it looks like a sheet of paper) on the left side of the window, and then navigate to the directory for this lab: `CC201 > labs > 1_ContainersAndDocker`. Click `Dockerfile` to view the commands required to build an image.



You can refresh your understanding of the commands mentioned in the Dockerfile below:

The FROM instruction initializes a new build stage and specifies the base image that subsequent instructions will build upon.

The COPY command enables us to copy files to our image.

The RUN instruction executes commands.

The EXPOSE instruction exposes a particular port with a specified protocol inside a Docker Container.

The CMD instruction provides a default for executing a container, or in other words, an executable that should run in your container.

3. Run the following command to build the image:

- docker build . -t myimage:v1

As seen in the module videos, the output creates a new layer for each instruction in the Dockerfile.

```
theia@theiadocker-[redacted]:/home/project/CC201/labs/1_ContainersAndDocker$ docker build . -t myimage:v1
Sending build context to Docker daemon  4.096kB
Step 1/6 : FROM node:9.4.0-alpine
9.4.0-alpine: Pulling from library/node
605ce1bd3f31: Pull complete
fe58b30348fe: Pull complete
46ef8987ccbd: Pull complete
Digest: sha256:9cd67a00ed111285460a83847720132204185e9321ec35dacec0d8b9bf674adf
Status: Downloaded newer image for node:9.4.0-alpine
 ---> b5f94997f35f
Step 2/6 : COPY app.js .
 ---> cced62775b60
Step 3/6 : COPY package.json .
 ---> 578384eb7c99
Step 4/6 : RUN npm install &&    apk update &&    apk upgrade
 ---> Running in 7f75ec5d9d5c
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN hello-world-demo@0.0.1 No repository field.
npm WARN hello-world-demo@0.0.1 No license field.

added 50 packages in 1.638s
fetch http://dl-cdn.alpinelinux.org/alpine/v3.6/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.6/community/x86_64/APKINDEX.tar.gz
v3.6.5-44-gda55e27396 [http://dl-cdn.alpinelinux.org/alpine/v3.6/main]
v3.6.5-34-gf0ba0b43d5 [http://dl-cdn.alpinelinux.org/alpine/v3.6/community]
OK: 8448 distinct packages available
Upgrading critical system libraries and apk-tools:
(1/1) Upgrading apk-tools (2.7.5-r0 -> 2.7.6-r0)
Executing busybox-1.26.2-r9.trigger
Continuing the upgrade transaction with new apk-tools:
(1/7) Upgrading musl (1.1.16-r14 -> 1.1.16-r15)
(2/7) Upgrading busybox (1.26.2-r9 -> 1.26.2-r11)
Executing busybox-1.26.2-r11.post-upgrade
(3/7) Upgrading libressl2.5-libcrypto (2.5.5-r0 -> 2.5.5-r2)
(4/7) Upgrading libressl2.5-libssl (2.5.5-r0 -> 2.5.5-r2)
(5/7) Installing libressl2.5-libtls (2.5.5-r2)
(6/7) Installing ssl_client (1.26.2-r11)
(7/7) Upgrading musl-utils (1.1.16-r14 -> 1.1.16-r15)
Executing busybox-1.26.2-r11.trigger
OK: 5 MiB in 15 packages
Removing intermediate container 7f75ec5d9d5c
 ---> abe7e7a3b349
Step 5/6 : EXPOSE  8080
 ---> Running in 26ad3df5ce52
Removing intermediate container 26ad3df5ce52
 ---> 44b98c2b942b
Step 6/6 : CMD node app.js
 ---> Running in bde00436d863
```

4. List images to see your image tagged myimage:v1 in the table.

- docker images

```
theia@theiadocker-[redacted]:/home/project/CC201/labs/1_ContainersAndDocker$ docker images
REPOSITORY    TAG            IMAGE ID       CREATED          SIZE
myimage       v1             cca37dd4d014   46 seconds ago   76.3MB
hello-world   latest         feb5d9fea6a5   6 months ago     13.3kB
node          9.4.0-alpine   b5f94997f35f   4 years ago      68MB
theia@theiadocker-[redacted]:/home/project/CC201/labs/1_ContainersAndDocker$
```

Note that compared to the hello-world image, this image has a different image ID. This means that the two images consist of different layers – in other words, they're not the same image.

You should also see a node image in the images output. This is because the docker build command pulled node:9.4.0-alpine to use it as the base image for the image you built.

Run the image as a container

1. Now that your image is built, run it as a container with the following command:

- docker run -dp 8080:8080 myimage:v1

```
theia@theiadocker-lavanyas:/home/project/CC201/labs/1_ContainersAndDocker$ docker run -dp 8080:8080 myimage:v1
1a8c245f482950cba52bcdb72686a8435e6c8916c6446434da55f5faac2372f3
theia@theiadocker-lavanyas:/home/project/CC201/labs/1_ContainersAndDocker$
```

The output is a unique code allocated by docker for the application you are running.

2. Run the curl command to ping the application as given below.

- curl localhost:8080

```
theia@theiadocker-lavanyas:/home/project/CC201/labs/1_ContainersAndDocker$ curl localhost:8080
Hello world from 1a8c245f4829! Your app is up and running!
```

If you see the output as above, it indicates that **'Your app is up and running!'.**

4. Now to stop the container we use docker stop followed by the container id. The following command uses docker ps -q to pass in the list of all running containers:

- docker stop $(docker ps -q)

```
theia@theiadocker-lavanyas:/home/project/CC201/labs/1_ContainersAndDocker$ docker stop $(docker ps -q)
1a8c245f4829
```

5. Check if the container has stopped by running the following command.