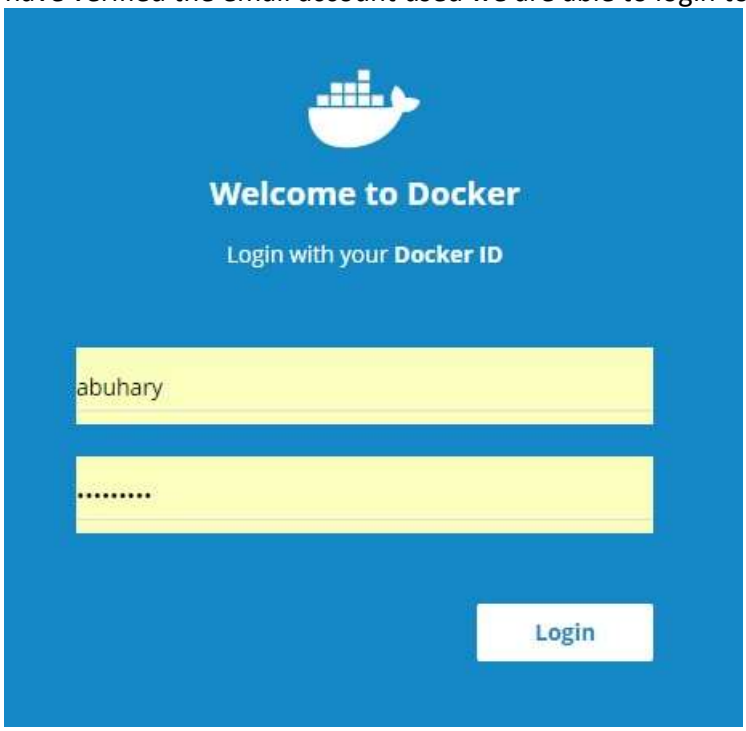
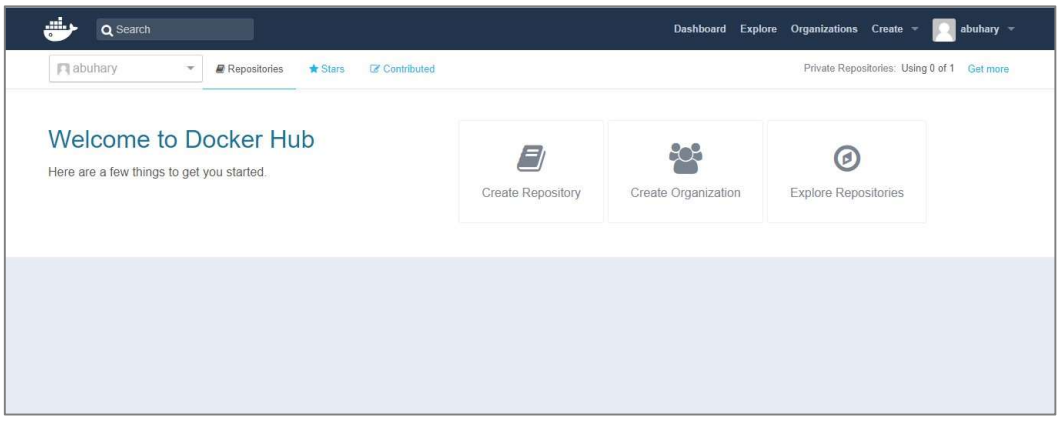
Deploying a Docker Container

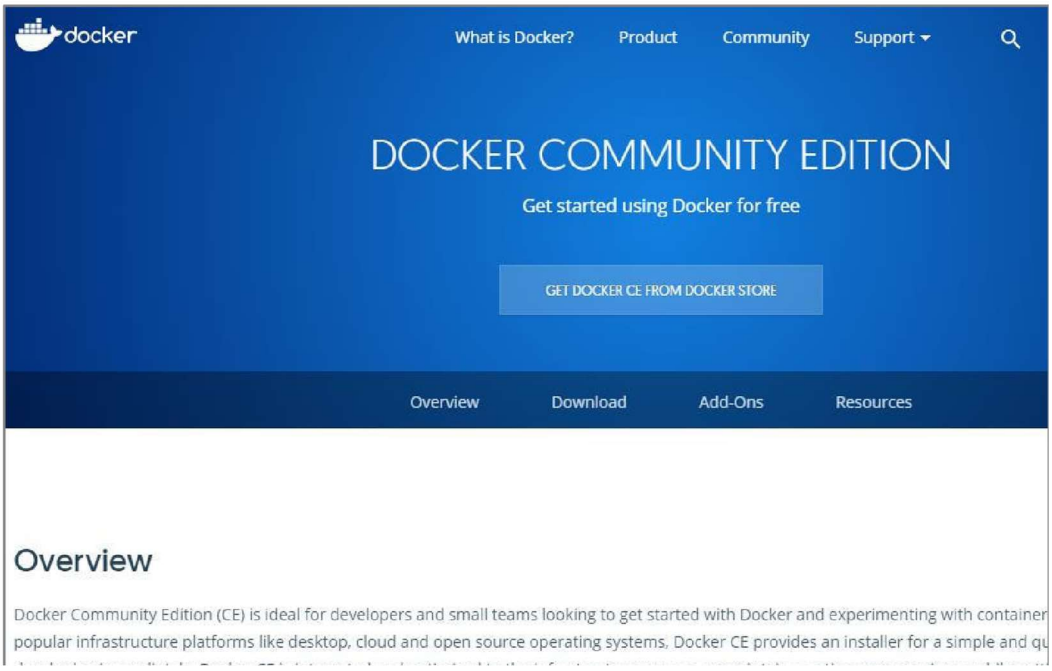
The first step is to go ahead and sign up for a free docker ID which you will use to create repositories and store all your containers and applications on. We head to https://hub.docker.com/ and enter the required information and create a Docker ID, once we have verified the email account used we are able to login to Docker Hub.



The dashboard presented can be used to create and view repositories and manage account settings.



Now that we have successfully created a Docker account we have to download and install the Docker client for windows. It is important to note that Docker in the past did not have a native application for Mac and windows which meant users had to download the Docker Tool-Box application which essentially used a virtual machine to run Docker on Linux. Docker now has a native application which can be downloaded directly for Mac and windows. The community edition of Docker is free to download and use from https://www.docker.com/community-edition



Using the Docker store, we are able to download the executable file which can then be used to install Docker on your local machine.

Once docker has been successfully installed it does not run automatically, search through the programs and launch docker. You will be able to tell when docker is up and running when you see the whale icon in the status bar staying still, and it can be easily accessed with the use of any terminal window.

In order to test if docker was successfully installed, open up any terminal window and run: **`docker --version`**. This should show the version of Docker that was installed on your computer.

```
> docker --version

Docker version 18.03.0-ce, build 0520e24
```

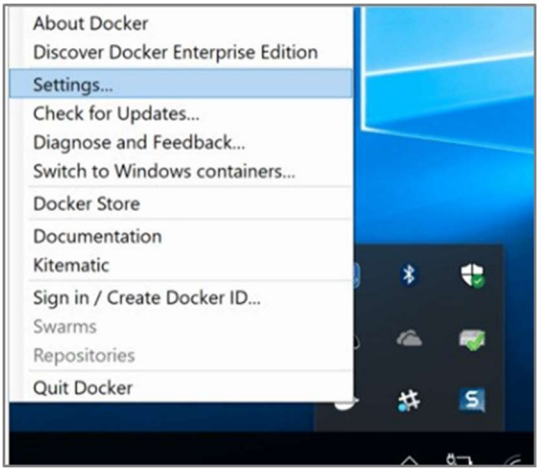
Hello World is an application that is already built and stored on the Docker Hub. To run that we simply type **`docker run hello-world`**.
Docker will try to run the app locally however as it realizes the file is not local, it will simply download the image from the hub and run it.

```
> docker run hello-world

docker : Unable to find image 'hello-world:latest' locally
...

latest:
Pulling from library/hello-world
ca4f61b1923c:
Pulling fs layer
ca4f61b1923c:
Download complete
ca4f61b1923c:
Pull complete
Digest: sha256:97ce6fa4b6cdc0790cda65fe7290b74cfebd9fa0c9b8c38e979330d547d22ce1
Status: Downloaded newer image for hello-world:latest

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

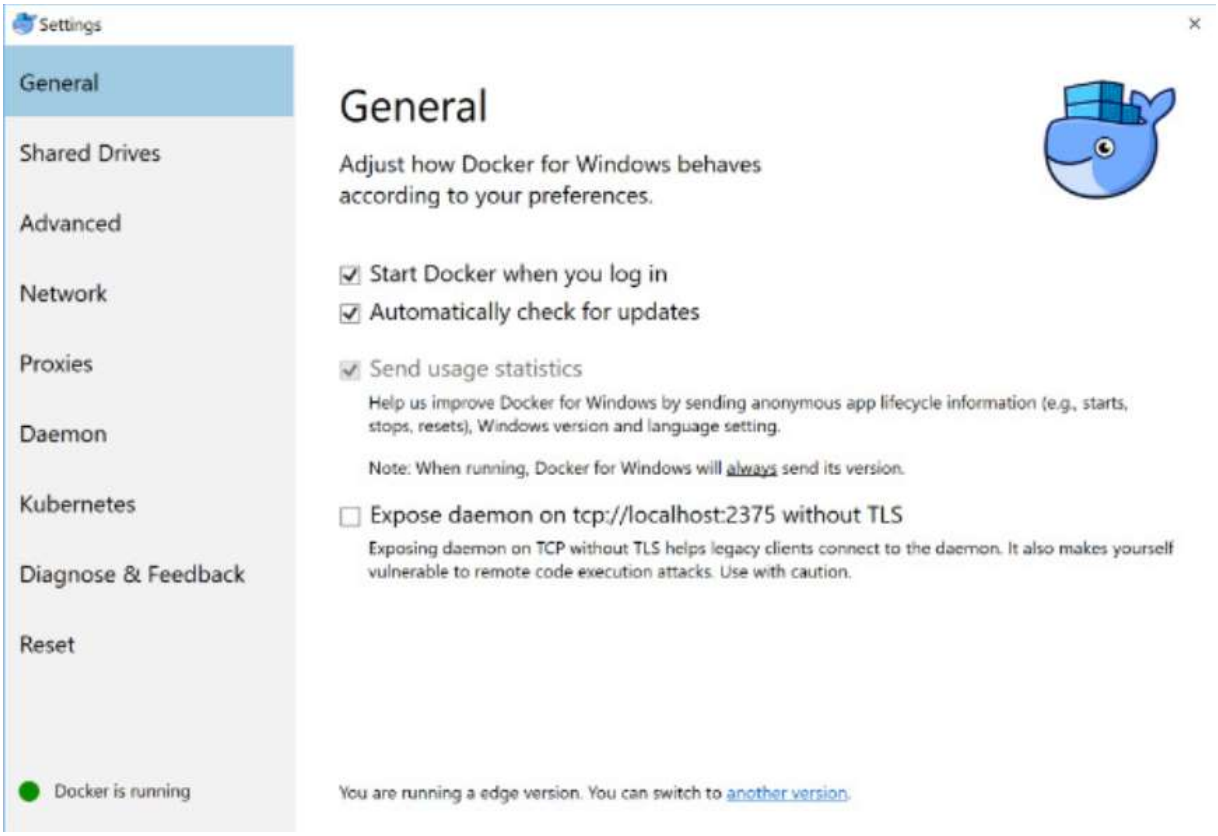
To view the container that was used to run the hello-world image we can run: **`docker container Is --all`**. We use the all command as the container automatically exits after displaying the message, it would not be needed if the container has not exited.

```
docker container ls --all

CONTAINER ID     IMAGE          COMMAND      CREATED             STATUS
54f4984ed6a8     hello-world    "/hello"     20 seconds ago      Exited (0) 19 seconds
```

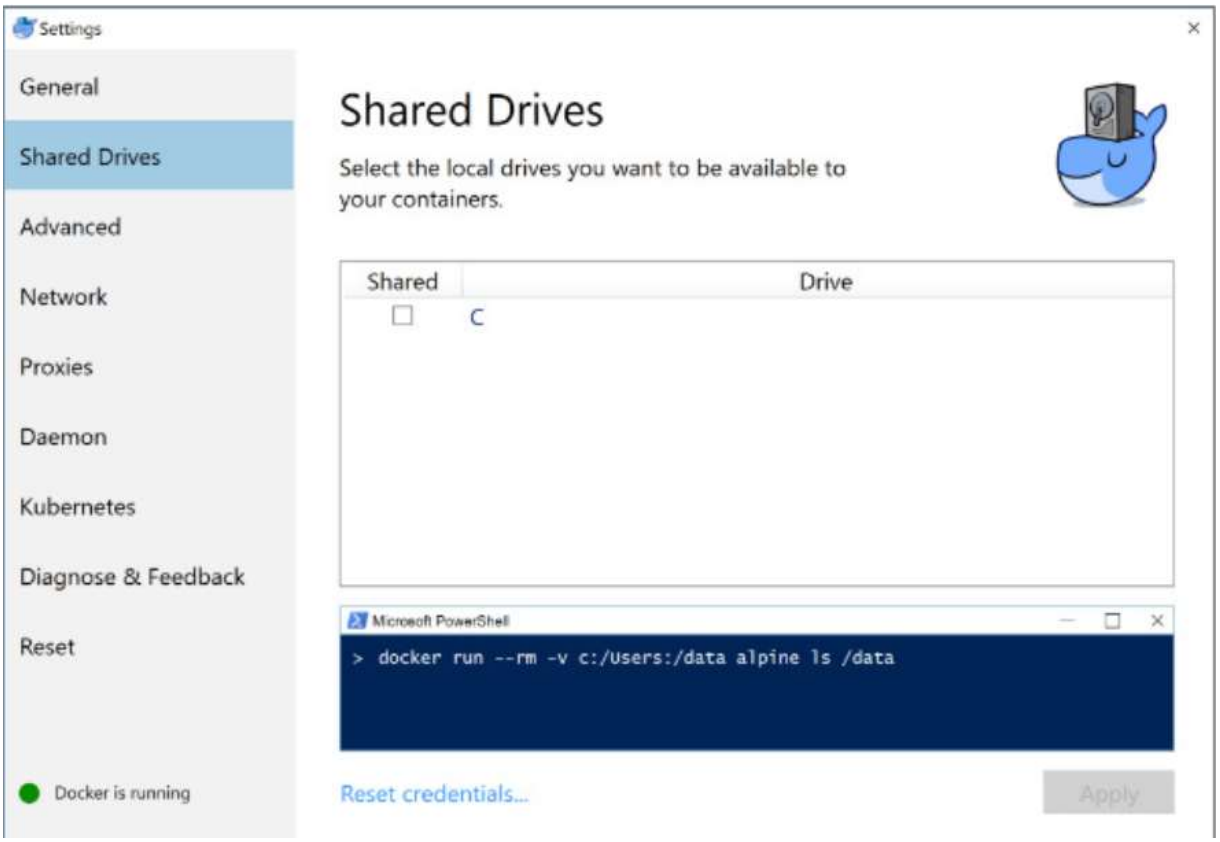
To configure Docker using specific settings, the Docker settings can be accessed by right-clicking the Docker icon in the status bar and selecting settings.



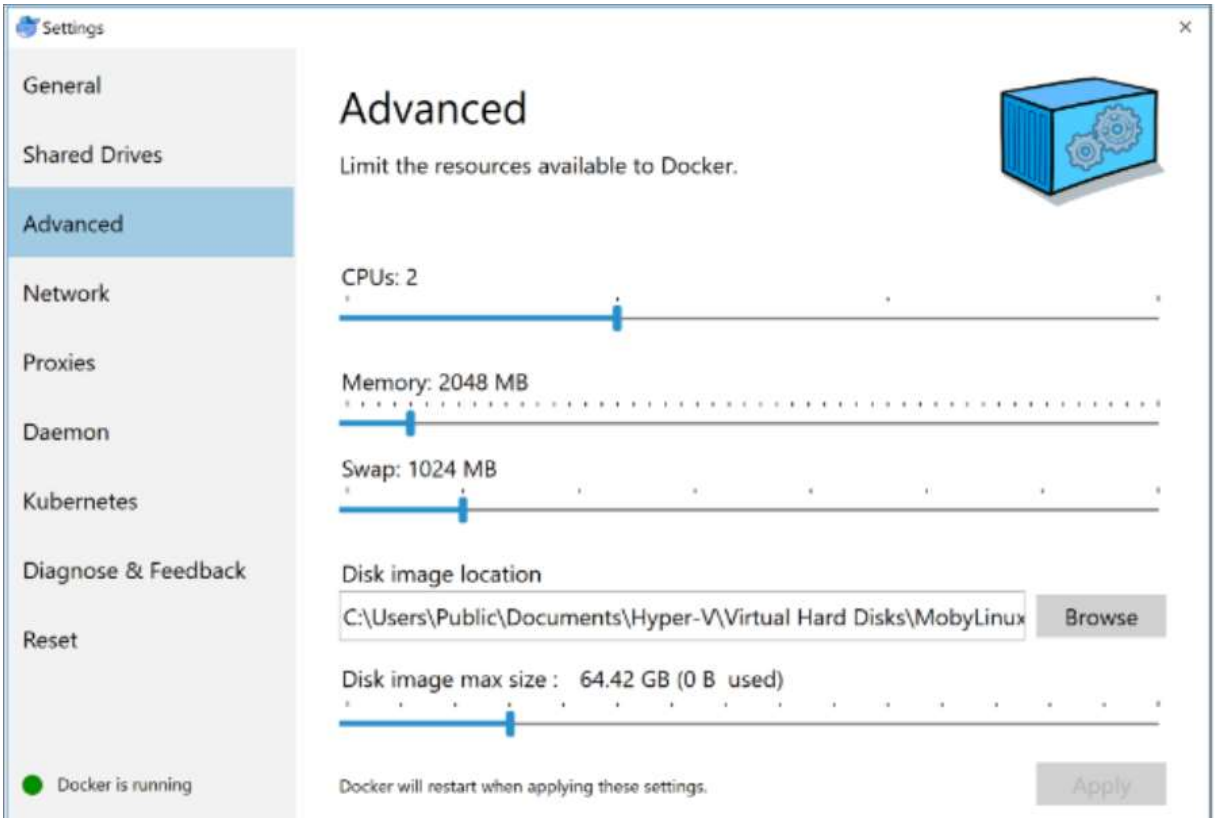
The general settings tab lets you select when to run and update Docker.



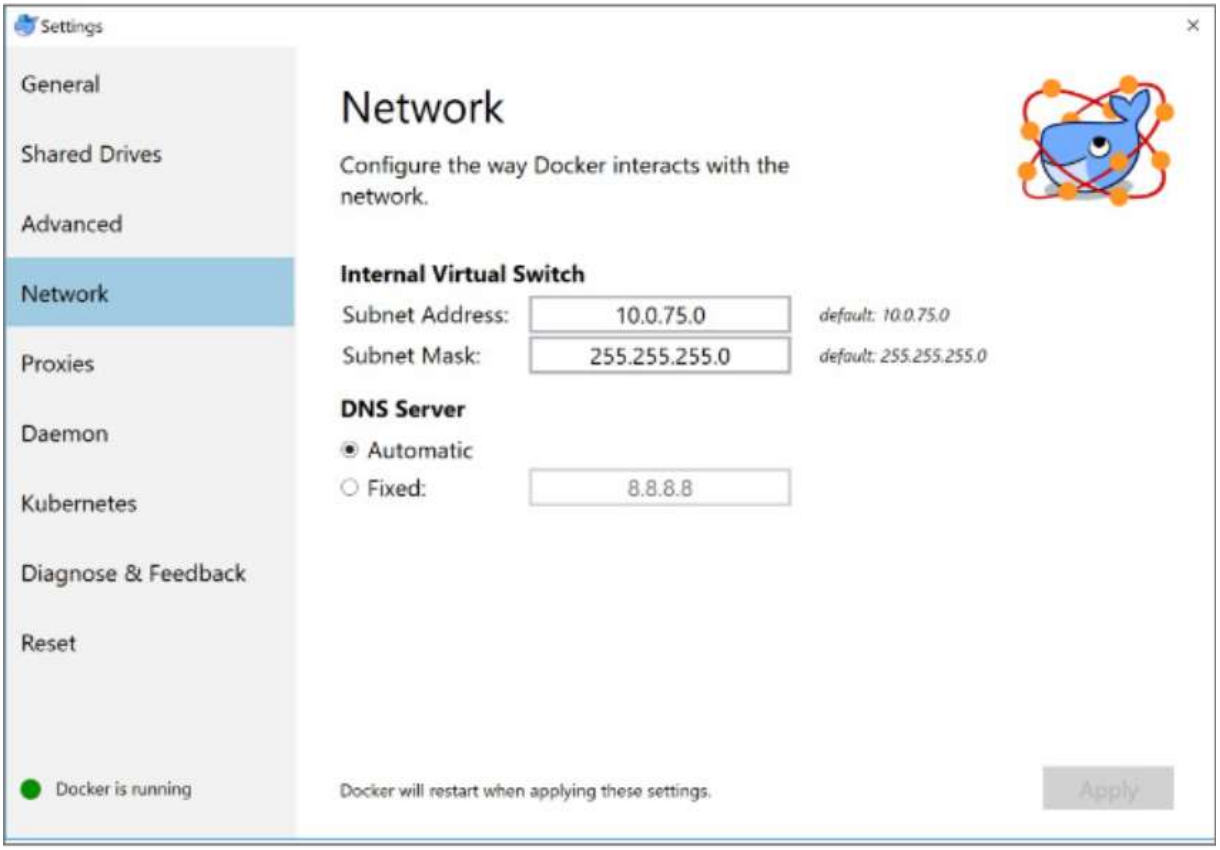
The shared drives section can be used to give Docker permission to make your local hard drives available to the Linux containers.



The advanced tab is used to set the number of processors and amount of memory assigned to the Linux VM.

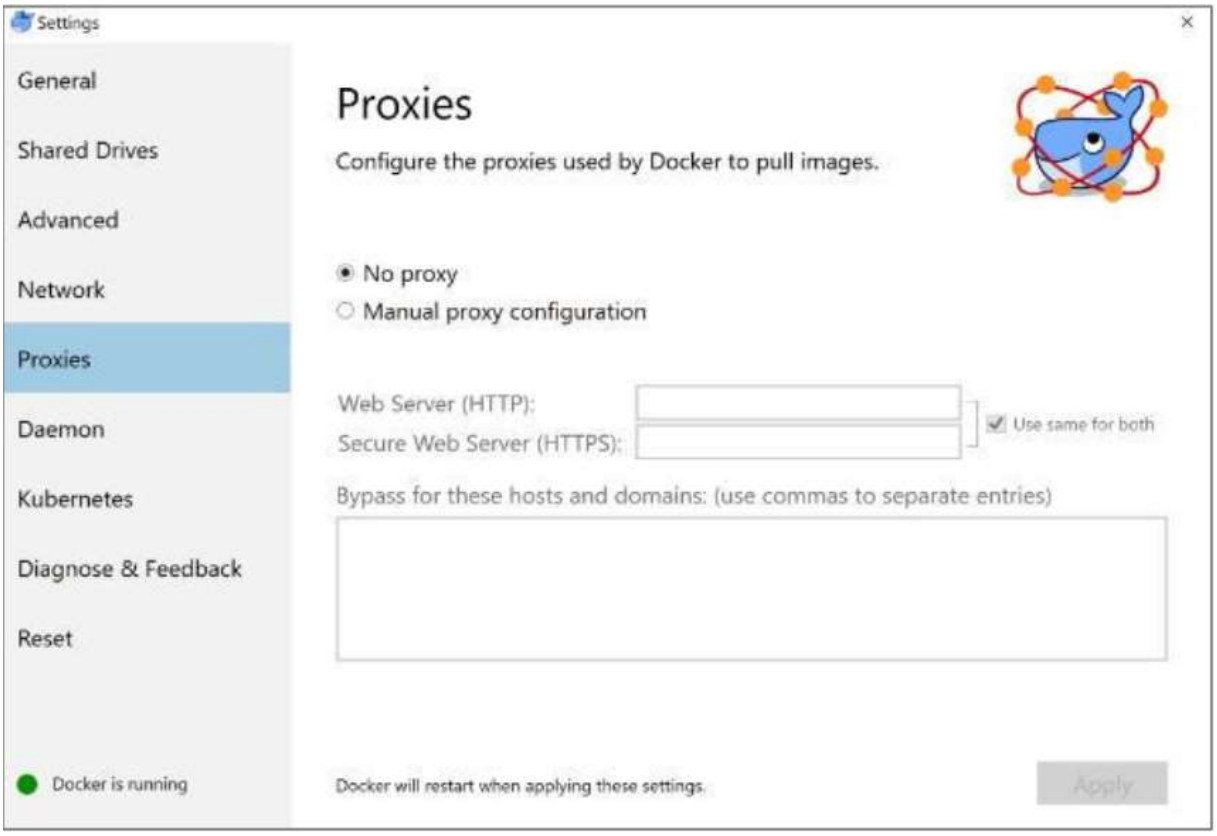


The network tab is used to configure Docker so that it runs on a Virtual Private Network (VPN).

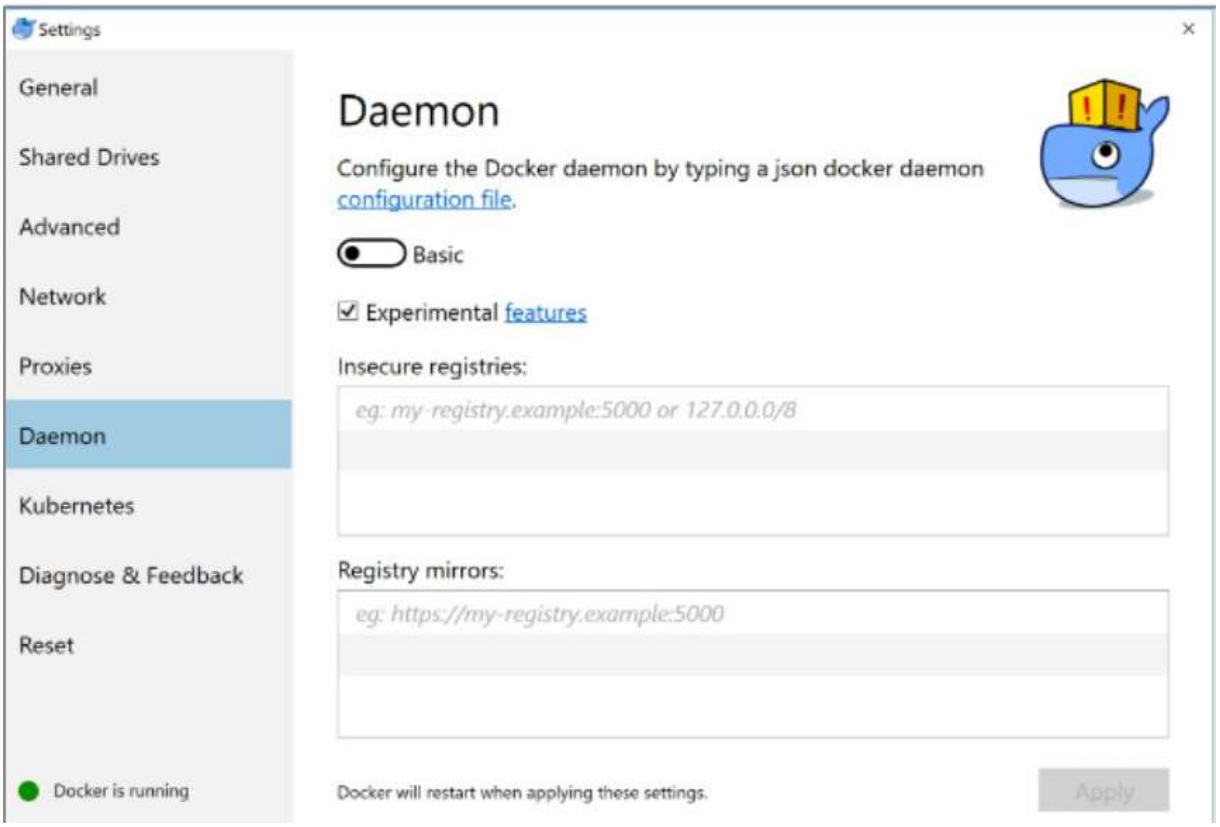


We are able to specify a network address translation (NAT) prefix and subnet mask to ensure internet accessibility. Docker also allows users to configure the DNS server to use a dynamic or static IP address.

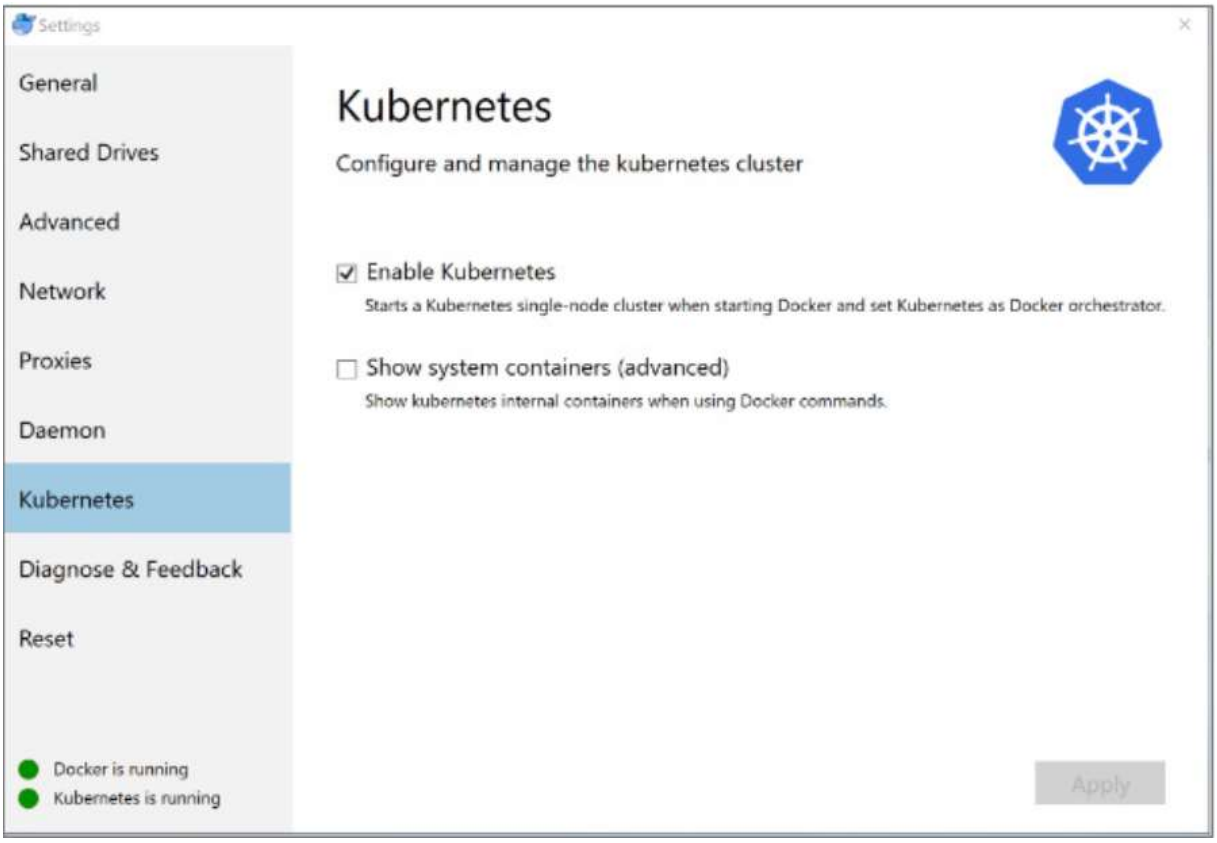
Use network settings to set up a VPN, and NAT for internet access. Can use DNS, static or dynamic IP addresses.



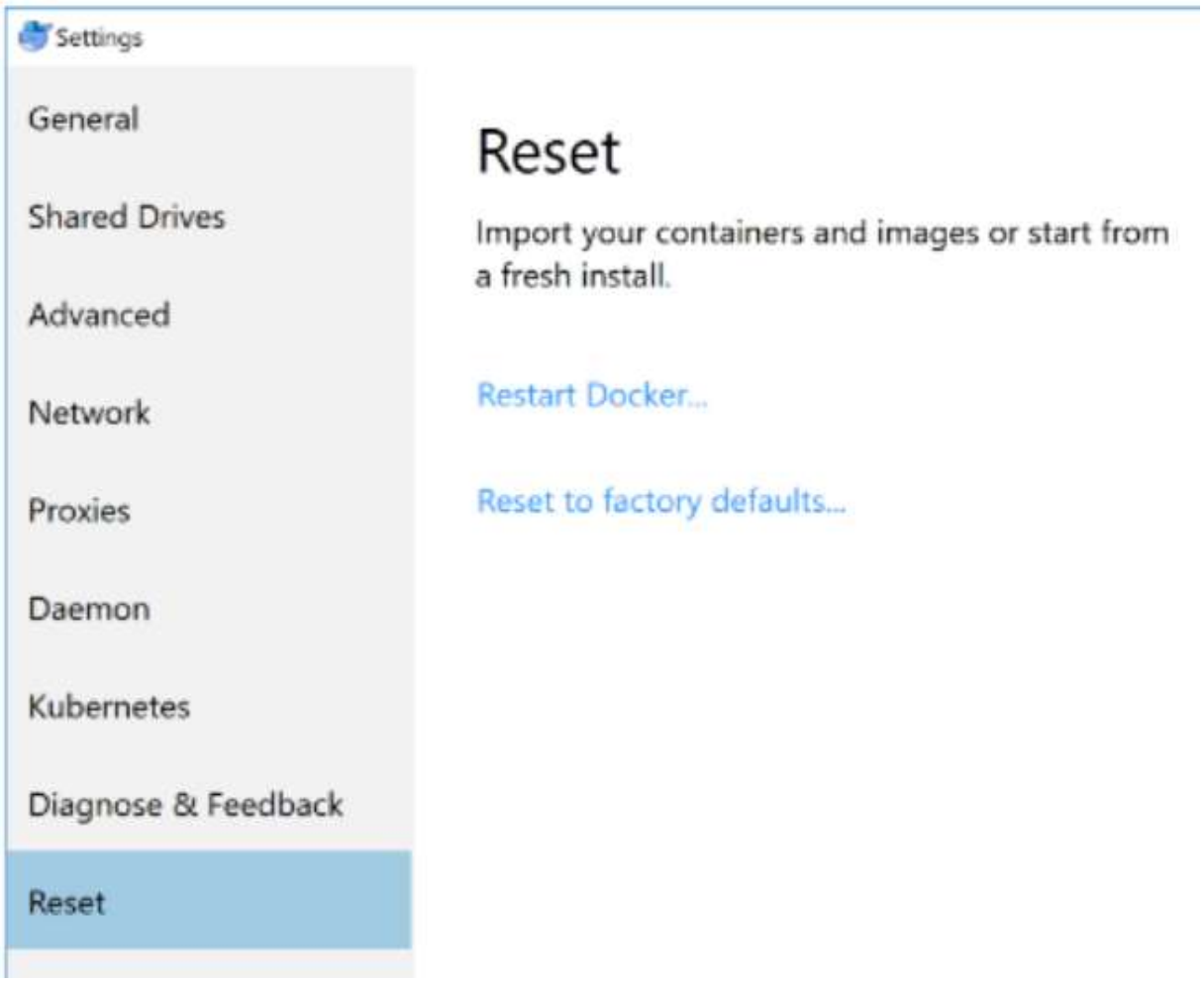
In the Proxies tab, Docker allows you to configure HTTP/HTTPS/ proxy settings and broadcast these to your containers. Docker will use the proxy set by the developer when pulling containers. Whenever changes are made to the proxy Docker automatically restarts to apply these new settings.



The Daemon tab lets you configure the Docker Daemon to hone how the containers are run in the environment. There are two modes available for developers; basic mode allows the configuration of common daemon options along with JSON and advanced mode, which allows you to directly edit JSON.



The Kubernetes tab allows users to enable the Kubernetes integration and test deploying to Kubernetes. However, Kubernetes is only functional on the Docker CE Edge version and not the standard version.



In the case that Docker becomes unresponsive or stops working the restart tab can be used to reset Docker back to its factory settings.

Now let's begin to write our own simple application and deploy it using a container. In the past, if you were to start developing a Python application, the first thing you would do is install the required Python runtime and development environments. However, this beats the whole purpose of portability and being able to develop apps without having environmental constraints.
Docker allows you to use a portable Python runtime image, meaning the base build will include the Python image along with the developer's code all packaged together in a container. A Dockerfile is a name that is given to these portable images that developers use to facilitate runtime requirements.
A Dockerfile is essentially like an environment configuration file, it defines the environment properties and access to resources such as networking interfaces is virtualized inside this file. As the resources are isolated from the rest of the system, appropriate ports need to be mapped and the developer has to define what files will be copied into the virtual

environment. Once it has been the Dockerfile ensures the containers behavior regardless of where it is run.
To create a Dockerfile we change into a new directory using cd and create a file called Dockerfile into which we copy the following code. The comments made in green should make it clear as to what each line of code is doing. The file refers to two files app.py and requirements.txt both of which haven't been created and will be done in the next step.

```
# 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
ADD . /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"]
```

Whilst in the same folder create two new files, requirements.txt and app.py. This is essentially the app, as the Dockerfile is built into an image, the Dockerfile's *ADD* command ensures both files are present in the image. The *EXPOSE* command in the Dockerfile ensures the output from app.py is available over HTTP. Type the lines below into the created files and save them.

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)
```

requirements.txt

```
Flask
Redis
```

The requirements.txt file sets up an environment with Python and Flask, building the image does not install Python on your local device but simply uses the Python Image to run the application.
To build the application we have to ensure we are in the top level of the new directory. Running **`ls`** should show the following:

```
$ ls
Dockerfile              app.py                  requirements.txt
```

We then use the `build` command, which creates the Docker image. To give the image a user- friendly name we use **`-t`** to tag the image.

```
docker build -t friendlyhello .
```

To see the images in the local Docker image registry we use the `image` command

```
$ docker image ls

REPOSITORY            TAG                 IMAGE ID
friendlyhello         latest              326387cea398
```

We then use the `run` command to run the app. While doing so we use `-p` to map the local machines port 4000 to the containers published port 80.

```
docker run -p 4000:80 friendlyhello
```

You should see a message that the application is being served by Python at `http://0.0.0.0:80`

However, this information is being released from inside the container, which contains information of the port mapping that took place when the image was built. Therefore the correct URL is `http://localhost:4000` When you go to that URL on a web browser you should be presented with the following.

To stop the container, we use **`docker container stop <container NAME or ID>`**. If you are unsure as to what your container is called simply use **`docker container ls`** to get a list of the running containers.

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED
1fa4ab2cf395        friendlyhello       "python app.py"     28 seconds ago
```

```
docker container stop 1fa4ab2cf395
```

To demonstrate the portability of containers and the purpose of them, we upload the image built and run the application elsewhere. The account you created at the start is used to create repositories and store images. The docker command line interface uses the public registry by default. We login to the Docker public registry using **`docker login`**

```
$ docker login
```

`username/repository:tag` is the notation used to associate a local image with a repository on the Docker registry. Developers use the tag to give the application some form of version control. To tag the image we run the following command:

```
docker tag image username/repository:tag
```

For example:

```
docker tag friendlyhello john/get-started:part2
```

To see your newly tagged docker image simply run **`docker image ls`**

```
$ docker image ls

REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
friendlyhello             latest              d9e555c53008        3 minutes ago       195MB
john/get-started          part2               d9e555c53008        3 minutes ago       195MB
python                    2.7-slim            1c7128a655f6        5 days ago          183MB
...
```

Once the image has been tagged, we have to push the image on to the registry using the **push** command.

```
docker push username/repository:tag
```

Once the image has been pushed to your account, you are able to pull and run it on any machine that contains docker. From any machine simply run:

```
docker run -p 4000:80 username/repository:tag
```

As the image is not available locally Docker will pull the image from the registry and run it.

```
$ docker run -p 4000:80 john/get-started:part2
Unable to find image 'john/get-started:part2' locally
part2: Pulling from john/get-started
10a267c67f42: Already exists
f68a39a6a5e4: Already exists
9beaffc0cf19: Already exists
3c1fe835fb6b: Already exists
4c9f1fa8fcb8: Already exists
ee7d8f576a14: Already exists
fbccdcced46e: Already exists
Digest: sha256:0601c866aab2adcc6498200efd0f754037e909e5fd42069adeff72d1e2439068
Status: Downloaded newer image for john/get-started:part2
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

Regardless of where the run command is executed, Docker pulls the image from the repository along with Python and all the other dependencies that are listed in the Requirements.txt file to run the application.