Microservices

Lab 3 – Container Packaging

Operating-system-level virtualization is a server virtualization method in which the kernel of an operating system allows the existence of multiple isolated user-space instances, instead of just one. These instances are typically called containers and, less often, software containers, virtualization engines (VEs), or jails. Containers look and feel like real servers from the point of view of the service within the container.

On Unix-like operating systems, particularly Linux, this technology can be seen as an advanced implementation of the standard chroot mechanism, namespaces on Linux. In addition to isolation mechanisms, the kernel often provides resource-management features to limit the impact of one container's activities on other containers, CGroups in Linux.

Docker is an open-source software system that automates the deployment of applications inside software containers. Containers abstract an application's operating environment from the underlying operating system. Containers eliminate the overhead and startup latency of a full virtual machine while preserving most of the isolation benefits.

Docker can package an application and its dependencies into an image, which can be used to launch a container on any Linux system. This enables flexibility and portability, allowing applications to run reliably across a number of Linux distributions with various configurations in a range of cloud settings. The recent Microsoft release of Windows 2016 has brought Docker containers for Windows to production as well.

In this lab we will migrate our microservice to a Docker container image to simplify running and deploying it.

1. Install Docker

Start your classroom virtual machine and log in as “user” with the password “user”. Launch a new terminal and run the following command to install Docker:

```
user@ubuntu:~$ wget -qO- https://get.docker.com/ | sh

# Executing docker install script, commit: 490beaa
+ sudo -E sh -c apt-get update -qq >/dev/null
+ sudo -E sh -c apt-get install -y -qq apt-transport-https ca-certificates curl software-properties-common
>/dev/null
+ sudo -E sh -c curl -fsSL "https://download.docker.com/linux/ubuntu/gpg" | apt-key add -qq - >/dev/null
+ sudo -E sh -c echo "deb [arch=amd64] https://download.docker.com/linux/ubuntu xenial edge" >
/etc/apt/sources.list.d/docker.list
+ [ ubuntu = debian ]
+ sudo -E sh -c apt-get update -qq >/dev/null
+ sudo -E sh -c apt-get install -y -qq docker-ce >/dev/null
+ sudo -E sh -c docker version
Client:
 Version:      17.09.0-ce
 API version:  1.32
 Go version:   go1.8.3
 Git commit:   afdb6d4
 Built:        Tue Sep 26 22:42:18 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.09.0-ce
 API version:  1.32 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   afdb6d4
 Built:        Tue Sep 26 22:40:56 2017
 OS/Arch:      linux/amd64
 Experimental: false
If you would like to use Docker as a non-root user, you should now consider
adding your user to the "docker" group with something like:

  sudo usermod -aG docker user

Remember that you will have to log out and back in for this to take effect!

WARNING: Adding a user to the "docker" group will grant the ability to run
         containers which can be used to obtain root privileges on the
         docker host.
         Refer to https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface
         for more information.
user@ubuntu:~$
```

If the install fails, try running `sudo apt-get update` and then retry.

Normal user accounts must use the `sudo` command to run command line tools as *root*. For our in-class purposes, eliminating the need for

© Copyright Tuesday, Feb 12, 2019, 8:48 AM RX-M LLC

`sudo` execution of the `docker` command will simplify our practice sessions. To make it possible to connect to the local Docker daemon domain socket without `sudo` we need to add our user id to the *docker* group. To add the *user* user to the *docker* group execute the following command:

```
user@ubuntu:~$ sudo usermod -a -G docker user
```

Now display the ID and Group IDs for *user*:

```
user@ubuntu:~$ id user
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),110(lxd),115(lpadmin),116(sambashare),999(docker)
```

As you can see from the `id user` command, the account *user* is now a member of the *docker* group. Now try running `id` without an account name:

```
user@ubuntu:~$ id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),110(lxd), 115(lpadmin),116(sambashare)
```

While the *docker* group was added to your group list, your login shell maintains the old groups. After updating your user groups you will need to restart your login shell to ensure the changes take effect. Reboot your system to complete the installation.

```
user@ubuntu:~$ sudo reboot
```

While a reboot is not completely necessary it verifies that your system is properly configured to boot up with the Docker daemon running.

When the system has restarted log back in as *user* with the password *user*.

2. Run a Test Container

To further test our system we will run a simple container. Use the `docker container run` subcommand to run the rx-m hello-world image:

```
user@ubuntu:~$ docker container run rxmllc/hello

Unable to find image 'rxmllc/hello:latest' locally
latest: Pulling from rxmllc/hello
9fb6c798fa41: Pull complete
3b61febd4aef: Pull complete
9d99b9777eb0: Pull complete
d010c8cf75d7: Pull complete
7fac07fb303e: Pull complete
5c9b4d01f863: Pull complete
Digest: sha256:0067dc15bd7573070d5590a0324c3b4e56c5238032023e962e38675443e6a7cb
Status: Downloaded newer image for rxmllc/hello:latest
 _____________________________________
/ RX-M - Cloud Native Consulting! \
\ rx-m.com                        /
 ---------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

user@ubuntu:~$
```

You now have a working Docker installation!

Food for thought:

- Where might you get more examples and ideas for using Docker?
- Did the Docker Engine run the container from a hello image it found locally?
- In the Docker output the hello image is listed with a "suffix", what is the suffix?
- What do you think this suffix represents?

3. Create a Dockerfile for your Inventory Microservice

Dockerfiles are usually placed at the root of a folder hierarchy containing everything you need to build one or more Docker images. This folder is called the build context. The entire build context will be packaged and sent to the Docker Engine for building. In simple cases the build context will have nothing in it but a Dockerfile.

Navigate to your inventory project directory:

© Copyright Tuesday, Feb 12, 2019, 8:48 AM RX-M LLC

```
user@ubuntu:~$ cd trash-can/

user@ubuntu:~/trash-can$ cd inv/
```

A Dockerfile contains a list of instructions for the Docker *build* subcommand to perform when creating the target image. We will build the following Dockerfile:

```
FROM ubuntu:16.04
RUN apt-get update -y && \
    apt-get install -y python-pip && \
    pip install flask && \
    rm -rf /var/lib/apt/lists/*
ENV FLASK_APP=inv.py
COPY . /app
WORKDIR /app
ENTRYPOINT ["python"]
CMD ["-m", "flask", "run", "-h", "0.0.0.0", "-p", "8080"]
EXPOSE 8080
```

Dockerfile statements can be loosely organized into three groups:

- **Filesystem Setup** – Steps that copy or install files into the image's file system (programs, libraries, etc.)
- **Image Metadata** – Descriptive image data outside of the filesystem (exposed ports, labels, etc.)
- **Execution Metadata** – Commands and environment settings that tell Docker what to do when users run the image.

Our Dockerfile has two statements in the **Image Metadata** category, FROM and EXPOSE. Dockerfiles begin with a FROM statement which defines the parent image for the new image. We will base our service on the "Ubuntu 16.04" image, just like the host we developed it on. You can create images with no parent by using the reserved name "scratch" as the parent image (e.g. FROM scratch). No matter where users run our container (RedHat EL, SUSE, etc.) inside the container, it will look like Ubuntu 16.04.

The EXPOSE instruction describes ports that the container's service(s) typically listen on, in our case 8080.

The **File System Setup** parts of our Dockerfile will need to run `apt-get` commands to update the system and `pip` commands to configure the necessary Python libraries. Note that because our service will now have its own private container, we do not need to bother with Virtual environments, otherwise the commands are identical to those we ran on the lab system earlier.

The RUN instruction is provided a string of commands to execute in sequence. This ensures that we update the package indexes (`apt-get update` generates about 30MB in index data) and clear them (`rm .../lists/*`) before committing the image each Dockerfile line creates. We also need to copy our Python program into the container image, the `/app` directory is used in our Dockerfile as the destination directory.

The **Execution Metadata** section of our Dockerfile will need to setup an environment variable for the Flask app (`inv.py`). This section also sets the working directory to `/app` and then runs `python` with the necessary parameters to launch our service on port 8080, listening on all interfaces inside the container (-h 0.0.0.0).

Create a Dockerfile for your service as per below:

```
user@ubuntu:~/trash-can/inv$ vim dockerfile

user@ubuntu:~/trash-can/inv$ cat dockerfile

FROM ubuntu:16.04
RUN apt-get update -y && \
    apt-get install -y python-pip && \
    pip install flask && \
    rm -rf /var/lib/apt/lists/*
ENV FLASK_APP=inv.py
COPY . /app
WORKDIR /app
ENTRYPOINT ["python"]
CMD ["-m", "flask", "run", "-h", "0.0.0.0", "-p", "8080"]
EXPOSE 8080
```

You can use the Dockerfile reference to look up any commands you would like more information on:
https://docs.docker.com/engine/reference/builder/

Before we build the Dockerfile we need to create a `.dockerignore` file. This file serves the same purpose as the `.gitignore` . If we do not tell Docker to ignore certain files it will copy them into our image when the COPY instruction runs. The only file we want to copy into our image is the `inv.py` file at present. Create the following `.dockeringnore` file:

```
user@ubuntu:~/trash-can/inv$ vim .dockerignore

user@ubuntu:~/trash-can/inv$ cat .dockerignore
```


© Copyright Tuesday, Feb 12, 2019, 8:48 AM RX-M LLC

```
venv/
.git/
.gitignore
dockerfile
.dockerignore
*.pyc
```

This will keep the COPY command from copying the `.git` and `venv` Python virtual environment into the container image.

Now build your Docker image:

```
user@ubuntu:~/trash-can/inv$ docker build -t trash-can/inv:0.1 ~/trash-can/inv/

Sending build context to Docker daemon  6.656kB
Step 1/8 : FROM ubuntu:16.04
16.04: Pulling from library/ubuntu
9fb6c798fa41: Pull complete
3b61febd4aef: Pull complete
9d99b9777eb0: Pull complete
d010c8cf75d7: Pull complete
7fac07fb303e: Pull complete
Digest: sha256:d45655633486615d164808b724b29406cb88e23d9c40ac3aaaa2d69e79e3bd5d
Status: Downloaded newer image for ubuntu:16.04
 ---> 2d696327ab2e
Step 2/8 : RUN apt-get update -y &&     apt-get install -y python-pip &&     pip install --upgrade pip &&     pip
install flask &&     rm -rf /var/lib/apt/lists/*
 ---> Running in cee148064fd3
...
 ---> 58602a8351c5
Removing intermediate container cee148064fd3
Step 3/8 : ENV FLASK_APP inv.py
 ---> Running in 247d37b14f23
 ---> 6082a7328783
Removing intermediate container 247d37b14f23
Step 4/8 : COPY . /app
 ---> 1720fe391275
Step 5/8 : WORKDIR /app
 ---> 1326199a326a
Removing intermediate container 27dc8f2e5a79
Step 6/8 : ENTRYPOINT python
 ---> Running in 93b404e5fa07
 ---> 40401be0d623
Removing intermediate container 93b404e5fa07
Step 7/8 : CMD -m flask run -h 0.0.0.0 -p 8080
 ---> Running in debc552ef3b9
 ---> 24dc5ea0b83f
Removing intermediate container debc552ef3b9
Step 8/8 : EXPOSE 8080
 ---> Running in 8dca0a06f8ad
 ---> 882b420bbbb7
Removing intermediate container 8dca0a06f8ad
Successfully built 882b420bbbb7
Successfully tagged trash-can/inv:0.1
user@ubuntu:~/trash-can/inv$
```

This will take some time, as the Docker base image for ubuntu:16.04 is fairly stripped down and many packages will need to be install to support `pip` and `flask` respectively.

Use the `docker image ls` subcommand to display the images on your lab system after the build completes:

```
user@ubuntu:~/trash-can/inv$ docker image ls

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
trash-can/inv       0.1                 882b420bbbb7        2 minutes ago       404MB
ubuntu              16.04               2d696327ab2e        3 weeks ago         122MB
rxmllc/hello        latest              05a3bd381fc2        3 weeks ago         1.84kB
user@ubuntu:~/trash-can/inv$
```

The three images are the hello-world image we ran to test Docker, the ubuntu image we based our service on and the *trash-can/inv* repository which contains the image we tagged 0.1 during the docker build.

Perfect.

4. Run the Containerized Service

To run our service we want to tell Docker to execute it in the background and to forward connections destined for port 8080 on the host

© Copyright Tuesday, Feb 12, 2019, 8:48 AM RX-M LLC

machine to the container. We should also give the container a name, use "inv". Launch your service with the following command:

```
user@ubuntu:~/trash-can/inv$ docker container run --name=inv -d -p=8080:8080 trash-can/inv:0.1

e35619c100b08948d4c75b9977fe56de8f3fe21f5f6605c322a28b95231ae950
user@ubuntu:~/trash-can/inv$
```

Use the `docker container ls` subcommand to view your container:

```
user@ubuntu:~/trash-can/inv$ docker container ls

CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
e35619c100b0        trash-can/inv:0.1    "python -m flask r..."   21 seconds ago      Up 20 seconds
0.0.0.0:8080->8080/tcp   inv
user@ubuntu:~/trash-can/inv$
```

The *container ls* subcommand shows our container running, displays the image we used and the command that it ran as well as the port mapping information.

You can use the `docker container logs` subcommand to view the console log:

```
user@ubuntu:~/trash-can/inv$ docker container logs inv

 * Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)

user@ubuntu:~/trash-can/inv$
```

Now try running a typical `curl` command to test the containerized microservice:

```
user@ubuntu:~/trash-can/inv$ curl -svv http://localhost:8080/

*   Trying ::1...
* Connected to localhost (::1) port 8080 (#0)
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.0
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: application/json
< Content-Length: 23
< Server: Werkzeug/0.12.2 Python/2.7.12
< Date: Tue, 10 Oct 2017 04:03:11 GMT
<
{
  "version": "0.1"
}
* Closing connection 0
user@ubuntu:~/trash-can/inv$
```

5. Clean Up and Commit

Now that we have an image built we can dispose of our microservice container without worries. We can run it again any time we like from the image. We can also save the image to a file on the host with `docker save`, or *push* the image to a registry server with `docker push`.

For now we'll simply stop and delete all containers:

```
user@ubuntu:~/trash-can/inv$ docker container rm $(docker container stop $(docker container ls -aq))

e35619c100b0
2eb632068254
user@ubuntu:~/trash-can/inv$
```

This command runs `docker container ls` in a sub shell and then forces the removal of all containers listed. The *-a* switch lists all containers, running or exited, and the *-q* switch outputs only the container IDs.

Now run the `git status` subcommand:

```
user@ubuntu:~/trash-can/inv$ git status

On branch master
```


© Copyright Tuesday, Feb 12, 2019, 8:48 AM RX-M LLC

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .dockerignore
        Dockerfile

nothing added to commit but untracked files present (use "git add" to track)
user@ubuntu:~/trash-can/inv$
```

We have yet to commit our `Dockerfile` and the `.dockerignore` file. Add both to the Git index and commit them to the inventory service source code repo, they are now a critical part of our microservice build:

```
user@ubuntu:~/trash-can/inv$ git add .dockerignore

user@ubuntu:~/trash-can/inv$ git add Dockerfile

user@ubuntu:~/trash-can/inv$ git commit -m "dockerfile and .dockerignore for trash can inventory service"

[master 5240d6f] dockerfile and .dockerignore for trash can inventory service
 2 files changed, 18 insertions(+)
 create mode 100644 .dockerignore
 create mode 100644 Dockerfile

user@ubuntu:~/trash-can/inv$ git status

On branch master
nothing to commit, working directory clean
```

Congratulation you have successfully completed the Lab!

Copyright (c) 2013-2018 RX-M LLC, Cloud Native Consulting, all rights reserved

© Copyright Tuesday, Feb 12, 2019, 8:48 AM RX-M LLC