RX-M

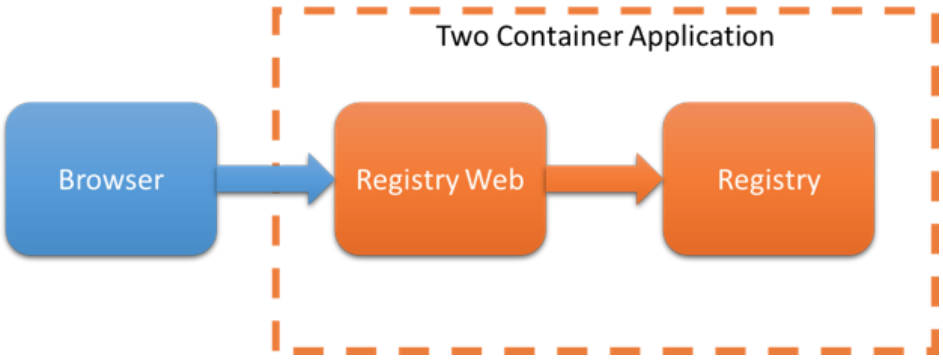
Docker In Practice

Lab – Networking

Docker makes it possible to deploy sets of containers as a unit. This is particularly useful in micro-service based systems where several services work together to provide the functionality required by an application. In this lab you will use container naming, linking and networking features to deploy an application consisting of multiple connected containers.

The application we will deploy provides a web browser view of a private registry server. We will need two services to support this application:

- Docker registry
- A web Frontend



We will use the hyper/docker-registry-web:v0.0.4 image to supply our web front end. We will use the docker registry:2 image to provide the Docker private registry service. To connect the two containers we will give the docker registry container a name and pass that name to the docker-registry-frontend.

1. Run a named private registry

Before you begin, make sure that there are no running containers on your lab system. If you still have containers running from a prior lab stop them with the `docker stop` command and then remove them with `docker rm`. Use `docker help` if you need help with any of these commands.

Now run a Docker registry container with its port mapped to the host and the container name reg-svr:

```
user@ubuntu:~$ docker run -p 5000:5000 --name reg_svr -d registry:2
f5ae45859abe5021ab5121ad14a01d38a3f3b7195a2051161a51fe204280216c

user@ubuntu:~$ docker ps
CONTAINER ID  IMAGE       COMMAND                  CREATED  STATUS  PORTS                    NAMES
f5ae45859abe  registry:2  "/bin/registry /etc/d"  13s ago  Up 12s  0.0.0.0:5000->5000/tcp  reg_svr
```

This is the first half of our application. Because we have named the container reg_svr we can locate this container from other containers using the container name.

2. Add a repository to the private registry

So that we have something to look at when we view our registry we will push a local image to the new registry. You can pick any local image to push to the new private registry server, the example below pulls busybox:latest for the purpose.

```
user@ubuntu:~$ docker pull busybox:latest
latest: Pulling from library/busybox
eeee0535bf3c: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:c1bc9b4bffe665bf014a305cc6cf3bca0e6effeb69d681d7a208ce741dad58e0
Status: Downloaded newer image for busybox:latest

user@ubuntu:~$ docker images busybox
REPOSITORY  TAG                 IMAGE ID            CREATED             SIZE
busybox     latest              0cb40641836c        4 weeks ago         1.114 MB
```

Create a new tag name for the image with the target registry server embedded in the repository part of the name:

```
user@ubuntu:~$ docker tag busybox:latest localhost:5000/bb:test

user@ubuntu:~$ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
registry                    2                   94d4bcbaa8d6        11 days ago         165.7 MB
busybox                     latest              0cb40641836c        4 weeks ago         1.114 MB
```

```
localhost:5000/bb            test                0cb40641836c        4 weeks ago         1.114 MB
...
```

Now push the image to your local registry server:

```
user@ubuntu:~$ docker push localhost:5000/bb:test
The push refers to a repository [localhost:5000/bb]
5f70bf18a086: Pushed
2c84284818d1: Pushed
test: digest: sha256:9176b113ee7620cb59f5df4c8afa86e990529e3ca328fe15cbc499eeb64bedfd size: 711
```

The example above pushes the busybox latest image to the localhost:5000 private registry with the repo name “bb” and the tag “test”.

3. Run the Web Interface

Now that we have the registry container running we can launch the web server for the registry. The web server needs to know where to find the registry server, however, every time you start the registry server it will be assigned a new IP address.

We can solve this problem with linking. We gave the registry server the name “reg_svr”, so we can now link the web server to the “reg_svr” container without knowing (or caring) what its IP address is. To run the web server container linked to the registry server container try the following:

```
user@ubuntu:~$ docker run -d \
                          -p 8080:8080 \
                          --link reg_svr:rs \
                          -e REGISTRY_HOST=rs \
                          -e REGISTRY_PORT=5000 \
                          hyper/docker-registry-web:v0.0.4
Unable to find image 'hyper/docker-registry-web:v0.0.4' locally
v0.0.4: Pulling from hyper/docker-registry-web
bbe1c4256df3: Pull complete
...
db29c1edd7e8: Pull complete
Digest: sha256:dc3ab1c87bccf19277d5f768bf5e6fef44cba85756f83f78beb8b38a47ba88b6
Status: Downloaded newer image for hyper/docker-registry-web:v.0.0.4
bf61ca78cc3e9e2fe4147056b5050f7e7753f997065adf08b0f9bb5bd6f84a1c
```

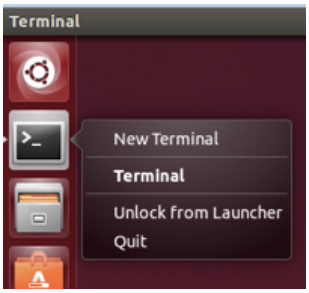
The command above launches the registry web server, linking the “rs” host name in the web server container to the IP address of the “reg_svr” container. We then use the -e switch to set the REGISTRY_HOST environment variable to the hostname the web server will use to reach the registry server.

4. Test the application

To test the linked container application we can use any browser that can reach our Lab VMs network interface.

- N.B. If you are using a cloud instance to complete the labs you will need to open port 8080 and browse to it over the internet.
- N.B. If your Lab VM is running on a desktop/laptop hypervisor and you can reach the Lab VM network interface from your desktop/laptop, you can use a browser from your desktop/laptop to view the application.

If you cannot browse to the Lab VM network interface externally you can install FireFox in the lab VM (this works fine but takes time to install and loads up the class room network with the FireFox download). To install FireFox on the Lab VM, right click the terminal icon in the launch pad and choose New Terminal to launch a second terminal:



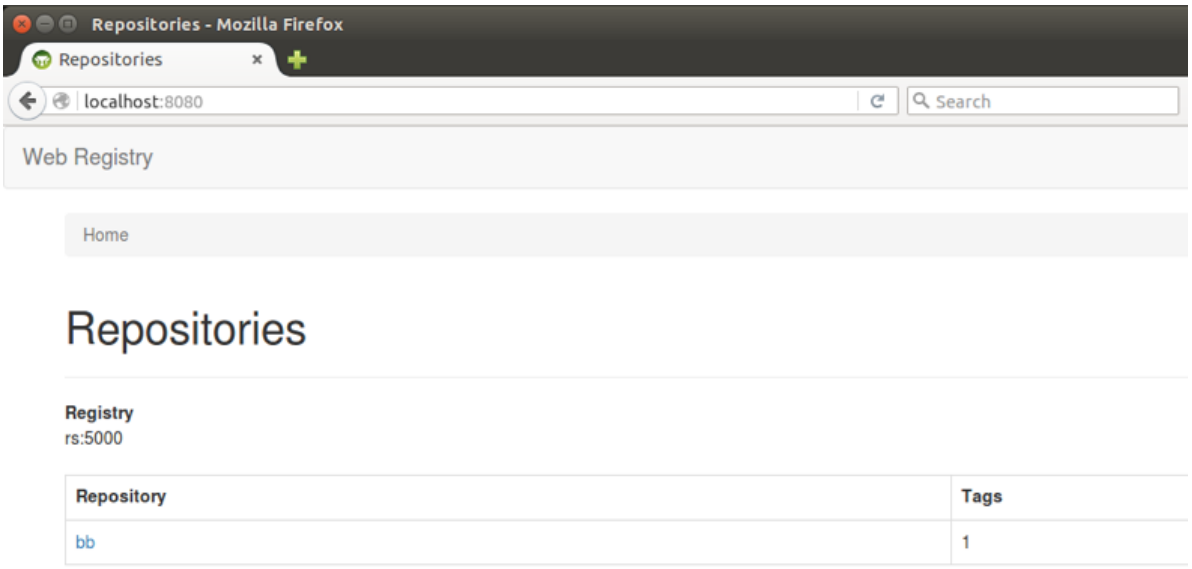
Use apt-get to install FireFox in the new terminal:

```
user@ubuntu:~$ sudo apt-get install -y firefox
...
```

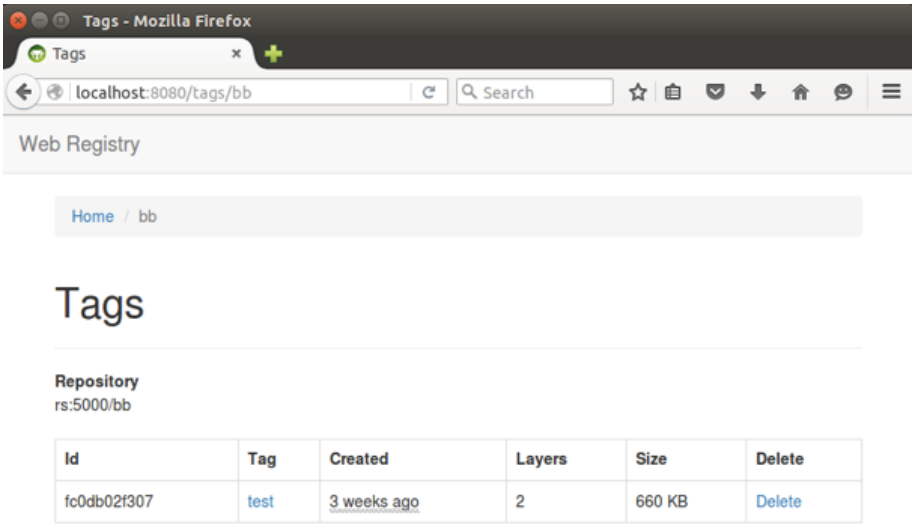
Once FireFox is installed, run the Firefox browser:

```
user@ubuntu:~$ firefox &
```

In the FireFox browser navigate to the web server’s mapped port http://localhost:8080.



Next click “bb” repository link to see the tagged images it contains.



This is a simple registry front end with only basic features, it does however provide a good example of container linking, as we now have a multi-container linked application running.

5. Monitor the application

We can monitor the status and health of our container services using various Docker commands.

Running containers

Use the `docker ps` subcommand to examine the services running in containers on your system:

```
user@ubuntu:~$ docker ps
CONTAINER ID  IMAGE                        COMMAND          CREATED   STATUS            PORTS                 NAMES
8a44f0e7974d  hyper/docker-registry-web:  catalina.sh run  1 hr ago  Up 1 hr  0.0.0.0:8080->8080/tcp  angry_rosalind
f384af0cb3fe  registry:2                   /bin/registry    1 hr ago  Up 1 hr  0.0.0.0:5000->5000/tcp  reg_svr
```

- Are the registry and web services up?
- If so for how long?

Container logs

View the logs of the two container services:

```
user@ubuntu:~$ docker logs --tail 5 reg_svr
172.17.0.3 - - [31/Dec/2015:00:15:17 +0000] "GET /v2/bb/manifests/test HTTP/1.1" 200 3373 "" "Java/1.7.0_79"
time="2015-12-31T00:15:18Z" level=info msg="response completed" go.version=go1.5.2 http.request.host="rs:5000" http.request.id=eec20a5b-b102-
4dee-9c71-d62dd1899c62 http.request.method=HEAD http.request.remoteaddr="172.17.0.3:35533"
http.request.uri="/v2/bb/blobs/sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4"
http.request.useragent="Java/1.7.0_79" http.response.contenttype="application/octet-stream" http.response.duration=1.896437ms
http.response.status=200 http.response.written=0 instance.id=3d6e649b-fb5f-4684-8e3f-67e0b4c2f346 version=v2.2.1
```

```
172.17.0.3 - - [31/Dec/2015:00:15:18 +0000] "HEAD /v2/bb/blobs/sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
HTTP/1.1" 200 0 "" "Java/1.7.0_79"
time="2015-12-31T00:15:18Z" level=info msg="response completed" go.version=go1.5.2 http.request.host="rs:5000" http.request.id=ce9a5abd-d391-
4ed0-962c-abb497434f0c http.request.method=HEAD http.request.remoteaddr="172.17.0.3:35533"
http.request.uri="/v2/bb/blobs/sha256:d7e8ec85c5abc60edf74bd4b8d68049350127e4102a084f22060f7321eac3586"
http.request.useragent="Java/1.7.0_79" http.response.contenttype="application/octet-stream" http.response.duration=1.270631ms
http.response.status=200 http.response.written=0 instance.id=3d6e649b-fb5f-4684-8e3f-67e0b4c2f346 version=v2.2.1
172.17.0.3 - - [31/Dec/2015:00:15:18 +0000] "HEAD /v2/bb/blobs/sha256:d7e8ec85c5abc60edf74bd4b8d68049350127e4102a084f22060f7321eac3586
HTTP/1.1" 200 0 "" "Java/1.7.0_79"

user@ubuntu:~$ docker logs 8a44f0e7974d
Dec 30, 2015 11:32:29 PM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["http-bio-8080"]
Dec 30, 2015 11:32:29 PM org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 490 ms
Dec 30, 2015 11:32:29 PM org.apache.catalina.core.StandardService startInternal
INFO: Starting service Catalina
Dec 30, 2015 11:32:29 PM org.apache.catalina.core.StandardEngine startInternal
INFO: Starting Servlet Engine: Apache Tomcat/7.0.52 (Ubuntu)
Dec 30, 2015 11:32:29 PM org.apache.catalina.startup.HostConfig deployWAR
INFO: Deploying web application archive /var/lib/tomcat7/webapps/ROOT.war
Dec 30, 2015 11:32:34 PM org.apache.catalina.startup.TaglibUriRule body
INFO: TLD skipped. URI: http://www.springframework.org/tags is already defined
Dec 30, 2015 11:32:34 PM org.apache.catalina.startup.TaglibUriRule body
INFO: TLD skipped. URI: http://www.springframework.org/tags/form is already defined
2015-12-30 23:32:43,307 [localhost-startStop-1] INFO  context.GrailsConfigUtils  - [GrailsContextLoader] Grails application loaded.
2015-12-30 23:32:43,376 [localhost-startStop-1] INFO  web.RestService  - Trying to connect http://rs:5000/v2
2015-12-30 23:32:43,483 [localhost-startStop-1] INFO  web.RestService  - HTTP status: 200
2015-12-30 23:32:43,483 [localhost-startStop-1] INFO  web.RestService  - Registry URL detected: http://rs:5000/v2
...
```

As you can see in the example above the web UI is accessing the registry server using the URI http://rs:5000. When we linked the web UI to the registry server, Docker injected the name “rs” and the IP address of the reg_svr into the web UI container’s /etc/hosts file.

Container Network Information

Inspect the IP address of the containers and the /etc/hosts file of the web UI container:

```
user@ubuntu:~$ docker inspect -f '{{.NetworkSettings.IPAddress}}' reg_svr
172.17.0.2

user@ubuntu:~$ docker inspect -f '{{.NetworkSettings.IPAddress}}' 8a44f0e7974d
172.17.0.3

user@ubuntu:~$ docker exec 8a44f0e7974d cat /etc/hosts
172.17.0.3    8a44f0e7974d
127.0.0.1    localhost
::1    localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
ff00::0    ip6-mcastprefix
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters
172.17.0.2    rs f384af0cb3fe reg_svr
```

When we launched the web UI container the --link reg_svr:rs switch added the “rs” hostname to the /etc/hosts file with the IP address of the registry container. This allowed us to tell the web UI to find the registry server using the hostname “rs”. Docker v1.10 provides DNS name resolution for containers on user defined networks.

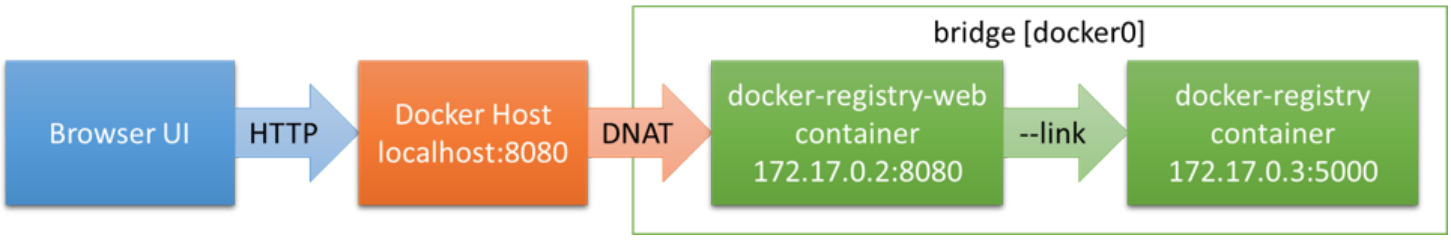
Both the registry server and the web ui container are on the default bridge network. Display the network information for the bridge network:

```
user@ubuntu:~$ docker network ls
NETWORK ID          NAME                DRIVER
8257d245c171        bridge              bridge
15a5d428d991        none                null
9aad8dd9b2b9        host                host

user@ubuntu:~$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "8257d245c1712859901a43deb1bfc2c52dcf645021af7d2168013523692f4698",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.17.0.0/16"
                }
```

```
            ]
        },
        "Containers": {
            "8a44f0e7974d926c0bee36dcd61a880b2c43c0cb41dfb34a460b3b071bd90a1c": {
                "EndpointID": "6a1a6990839fd5202e79443b77c4f246c852f0a419e3a45bf66660cf6898e870",
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            },
            "f384af0cb3fe5512adbfacafd3bc987aaed762227200f9567f138dab698aea7d": {
                "EndpointID": "b2d4724c8c145048ad5d27be6a7f90c71114d34c1fd6a2f727bf015d1fdd81a2",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        }
    }
]
```


The application as deployed in the example looks something like this:



The --link switch was the only “Docker supplied” way for one container to discover another by host name prior to Docker v1.9. Docker v1.9 introduced container networking, adding all named containers on a user defined network to the /etc/hosts file of other containers. In Docker v1.10 the Docker daemon acts as a DNS server allowing the same lookups without modifications to container /etc/hosts files.

We could rebuild the example above in Docker v1.9+ by creating a private network for the web UI and the registry server to share [e.g. `docker network create regnet`]. We could then connect both containers to the new network [e.g. `docker network connect regnet reg_svr`.] The --link switch is still supported in Docker v1.10+ and today acts like an alias command making it possible to lookup container IPs using the container name or the link name.

Container Filesystem Changes

What if we want to see where the registry server is storing images that we push to it? Try the `docker diff` subcommand:

```
user@ubuntu:~$ docker diff reg_svr
C /var
C /var/lib
A /var/lib/registry
```

The Docker registry service added the /var/lib/registry directory within the container. Because this is a new directory, it exists within the container only and files created under it will not show in the `docker diff`. This is the directory where docker-registry saves all of the images pushed to the server.

Container Runtime Information

What if we want to monitor the cpu and memory consumption of our services? Run the `docker stats` subcommand in a new shell to monitor the desired running containers (substitute your own container names):

```
user@ubuntu:~$ docker stats reg_svr angry_rosalind
```

This command will produce a continuously updating text table with various stats from both containers specified.

```
CONTAINER           CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O
angry_rosalind      0.22%               690.6 MB / 4.142 GB   16.67%              100.4 kB / 152 kB   0 B / 20.48 kB
reg_svr             0.01%               7.442 MB / 4.142 GB   0.18%               725.3 kB / 97.52 kB 0 B / 0 B
```

Use ^C (control + c) to terminate the stats display.

Next use the docker top command to display the processes running in the registry server container:

```
user@ubuntu:~$ docker top reg_svr
UID    PID    PPID   C    STIME    TTY    TIME        CMD
root   16405  947    0    15:25    ?      00:00:01    /bin/registry /etc/docker/registry/config.yml
```

Note that docker top shows the host based process ID of processes running in the container's PID namespace. Within the container the startup process is always PID 1:

```
user@ubuntu:~$ docker exec reg_svr ps -e
PID   USER     TIME   COMMAND
    1 root       0:01 registry serve /etc/docker/registry/config.yml
   13 root       0:00 ps -e
```

Congratulations! You have completed the Docker networking lab!!