Microservices

Lab 6 - Stateful Microservices

Applications composed of microservices contain both stateless and stateful services. It is important to understand the constraints and limitations of implementing stateful services. If a service relies on state, it should be separated into a dedicated container that's easily accessible.

One of the key advantages of microservices is their ability to scale rapidly. Like other distributed computing architectures, microservices scale better when they are stateless. Within seconds, multiple containers can be launched across multiple hosts. Each container implementing a given service is autonomous and has no knowledge of other instances of the service. Containerized microservices can be scaled precisely rather than scaling entire VMs hosting multiple services. For this pattern to work seamlessly, services should be stateless. Ephemeral container designs are an ideal choice for microservices.

A microservice based application will inevitably require stateful services as well. These are typically packaged as containers with unique attributes. Stateful services are typically backed by a data store which uses a volume and drivers such as DRBD, Blockbridge, etc. to make the persistence layer host independent. In many cases state can be offloaded to highly available cloud data stores to provide persistence. Cluster friendly data stores such as Redis, Cassandra, and Cloudant, maximize availability with minimal delay on consistency.

In this lab we will turn our trash level report service into a stateful container backed by persistent/durable Redis key value store.

1. Run a Redis Container with a Docker Volume

So far all of our containers are tracking state in memory. This is great for cashing but not durability. If we want to make our state durable (such that the state survives a container restart) we need to use a stateful container pattern. With Docker this usually means running a suitable data store for the data we have and mapping the container's storage directory to a volume.

The `docker volume` subcommand allows you to create storage volumes for Docker containers. These volumes have a lifespan independent of the containers that use them. Volumes can be created and deleted at anytime independent of the container(s) using the volume. Volume drivers allow volumes to be backed by storage arrays (EMC, HPE, etc.) or network filesystems (NFS, GlusterFS, etc.) or iSCSI/FC style volumes (OpenStack Cinder, AWS EBS, etc.).

Use the `docker volume` subcommand to create a volume for your trash level report data:

```
user@ubuntu:~/trash-can/report$ docker volume create --name report-data

report-data

user@ubuntu:~/trash-can/report$ docker volume ls

DRIVER              VOLUME NAME
local               report-data

user@ubuntu:~/trash-can/report$ docker volume inspect report-data

[
    {
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/report-data/_data",
        "Name": "report-data",
        "Options": {},
        "Scope": "local"
    }
]
user@ubuntu:~/trash-can/report$
```

Now we can run a Redis key/value store and tell it to persist its data to the volume:

```
user@ubuntu:~/trash-can/report$ docker container run --name rep-data -d --net iot-net -v report-data:/data redis
redis-server --appendonly yes

Unable to find image 'redis:latest' locally
latest: Pulling from library/redis
693502eb7dfb: Already exists
338a71333959: Pull complete
83f12ff60ff1: Pull complete
4b7726832aec: Pull complete
19a7e34366a6: Pull complete
622732cddc34: Pull complete
3b281f2bcae3: Pull complete
Digest: sha256:4c8fb09e8d634ab823b1c125e64f0e1ceaf216025aa38283ea1b42997f1e8059
```

```
Status: Downloaded newer image for redis:latest
9f1496b4e8a03029d43d0be20f640d2cf9415b6c7cb63225080be0eaf44ac73f
user@ubuntu:~/trash-can/report$
```

You can display the console output of the service using `docker container logs`:

```
user@ubuntu:~/trash-can/report$ docker container logs -f rep-data


                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 3.2.8 (00000000/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in standalone mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
      |    `-._   `._    /     _.-'    |     PID: 1
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |           http://redis.io
       `-._    `-._`-.__.-'_.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |
       `-._    `-._`-.__.-'_.-'    _.-'
           `-._    `-.__.-'    _.-'
               `-._        _.-'
                   `-.__.-'

1:M 22 Mar 10:51:13.844 # WARNING: The TCP backlog setting of 511 cannot be enforced because
/proc/sys/net/core/somaxconn is set to the lower value of 128.
1:M 22 Mar 10:51:13.845 # Server started, Redis version 3.2.8
1:M 22 Mar 10:51:13.845 # WARNING overcommit_memory is set to 0! Background save may fail under low memory
condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command
'sysctl vm.overcommit_memory=1' for this to take effect.
1:M 22 Mar 10:51:13.845 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will
create latency and memory usage issues with Redis. To fix this issue run the command 'echo never >
/sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the
setting after a reboot. Redis must be restarted after THP is disabled.
1:M 22 Mar 10:51:13.845 * The server is now ready to accept connections on port 6379
```

Looks good! Redis gives us some tuning warnings but it is ready to receive connections. The logs *-f* switch follows the log so we can monitor the Redis activity.

2. Update the Report Service

Now we need to update our trash level report service to save its state in the Redis k/v store. This way if the report server crashes we can simply restart it somewhere else and reattach it to the Redis container as if nothing ever happened. If Redis itself fails we can restart it somewhere else and simply attach it to the same volume.

In production, to protect the volume, we can use an HA driver that stores the volume on a RAID storage array that auto replicates data across regions (EMC and various others supply such features).

In another terminal, update your report server code:

```
user@ubuntu:~/trash-can/report$ vim rep.js

user@ubuntu:~/trash-can/report$ cat rep.js
```

```
var nats = require('nats').connect({'servers':['nats:iot-msg:4222']});
var express = require('express');
var http = require('http');
var redis = require('redis');

var app = express();
var client = redis.createClient('6379', 'rep-data');

nats.subscribe('trash-level', function(msg) {
    var rep = JSON.parse(msg);
    client.set(rep.can_id, rep.level);
    console.log('Saved report: ' + msg);
});

app.get('/reports/:can_id', function(req, res) {
    client.get(req.params.can_id, function(err, reply) {
        if (reply==null) {
            res.statusCode = 404;
```

```
            return res.send('Error: Trash Can not found\n');
        }
        console.log(reply);
        res.send('{"can_id":"' + req.params.can_id + '", "level":"'
        + reply + '"}');
    });
});

http.createServer(app).listen(9090, function() {
    console.log('Listening on port 9090');
});
```

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

Our new server uses four libraries:

- nats - to receive trash level messages from our IoT trash can network
- express - a nodeJS framework for building RESTful services
- http - used to create the web server our REST API will listen to
- redis - the library that gives us access to the Redis key/value store

The NATS code is as before but now we save all of the received trash levels in the Redis DB before we log them. The main block of code associated with the `app.get()` function is our express web services. We accept requests to the `/reports` IRI for a given can_id, returning that can's level.

This microservice is now following the CQRS pattern. Asynchronous messages (Commands) are received to update data and Synchronous REST calls are used to retrieve data. The CQRS pattern has several advantages but "read your own writes" is not one of them. If a client rapidly writes a trash level and then tries to read it they may not see the new trash level until it has made its way through the queue. In our application this is fine because the trash cans sending level updates never read, a perfect fir for CQRS.

Before proceeding make sure to start your NATS broker and simulator (if they are not still running from the previous lab).

In the example below the NATS broker (iot-msg) is running but the sim needs to be restarted. Telling docker to start a container that is already running is harmless:

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

CONTAINER ID  IMAGE    COMMAND                  CREATED     STATUS                     PORTS
NAMES
9f1496b4e8a0  redis    "docker-entrypoint..."   5 min ago   Up 5 minutes               6379/tcp
rep-data
26a64b293469  node     "/bin/bash"              10 min ago  Exited (130) 8 minutes ago
rep
5eb903eacb10  node     "/bin/bash"              14 min ago  Exited (130) 8 minutes ago
sim
0f274d4e98bd  busybox  "cat /etc/resolv.conf"   15 min ago  Exited (0) 15 minutes ago
keen_carson
1b32864dd038  busybox  "ping -c 1 iot-msg"      16 min ago  Exited (0) 16 minutes ago
zealous_khorana
44a6070943a8  nats     "/gnatsd -c gnatsd..."   19 min ago  Up 19 minutes              4222/tcp, 6222/tcp,
8222/tcp   iot-msg

user@ubuntu:~/trash-can/report$ docker start iot-msg

iot-msg

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

Docker does not delete containers on container exit. So although in this example the sim container is not displayed it is still on the drive and can be quickly restarted.

```
user@ubuntu:~/trash-can/report$ docker container ls -a -f name=sim

CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS
NAMES
5eb903eacb10        node                "/bin/bash"         15 minutes ago      Exited (130) 9 minutes ago
sim
user@ubuntu:~/trash-can/report$

user@ubuntu:~/trash-can/report$ docker container start sim

sim
```

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

CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
9f1496b4e8a0        redis               "docker-entrypoint..."   7 minutes ago       Up 6 minutes        6379/tcp
rep-data
5eb903eacb10        node                "/bin/bash"              16 minutes ago      Up 5 seconds
sim
44a6070943a8        nats                "/gnatsd -c gnatsd..."   21 minutes ago      Up 21 minutes       4222/tcp,
6222/tcp, 8222/tcp   iot-msg

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

We can reattach our terminal to the tty in the sim container using the Docker `attach` command. This will allow us to rerun the simulator program (press enter if the prompt does not appear immediately):

```
user@ubuntu:~/trash-can/report$ docker container attach sim

root@5eb903eacb10:/#

root@5eb903eacb10:/# node /app/sim.js

publishing: {"can_id":"187", "level":"69.38445994297054"}
publishing: {"can_id":"921", "level":"68.66595960969126"}
...
```

Now we're ready to run our new trash level reporting service. To run the new code we need to start a Node container and install all of the required libraries using the Node Package Manager (NPM). Then we can run our app. Open a new terminal and start the trash level reporting service:

```
user@ubuntu:~/trash-can/report$ docker container run -it -v ~/trash-can/report:/app --net=iot-net --name=rep-svr
node /bin/bash

root@90d7e2c9bea9:/# npm install nats
...

root@90d7e2c9bea9:/# npm install express
...

root@90d7e2c9bea9:/# npm install redis
...

root@90d7e2c9bea9:/# node /app/rep.js

Listening on port 9090
Saved report: {"can_id":"239", "level":"75.07291677741993"}
Saved report: {"can_id":"947", "level":"14.54091585210191"}
Saved report: {"can_id":"580", "level":"53.91708208574717"}
Saved report: {"can_id":"81", "level":"52.01598882368821"}
...
```

Great our new report server is up, receiving NATS trash level messages and saving them in the stateful Redis container.

3. Test Your New Service

Now we can use `curl` to test the report service. In a new terminal, list the containers running:

```
user@ubuntu:~$ docker container ls

CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
90d7e2c9bea9        node                "/bin/bash"              2 minutes ago       Up 2 minutes
rep-svr
9f1496b4e8a0        redis               "docker-entrypoint..."   10 minutes ago      Up 10 minutes       6379/tcp
rep-data
5eb903eacb10        node                "/bin/bash"              19 minutes ago      Up 4 minutes
sim
44a6070943a8        nats                "/gnatsd -c gnatsd..."   25 minutes ago      Up 25 minutes       4222/tcp,
6222/tcp, 8222/tcp   iot-msg

user@ubuntu:~$
```

Our NATS server is running to handle message distribution (iot-msg,) our simulator is running the node image, our redis container is running

(rep-data) and our report server is running the node image as well. In a practical setting we would create Dockerfiles for the simulator and report server and build them into images that we could just run, but our ad hoc environment works fine for testing and development.

Now discover the IP address of your report server:

```
user@ubuntu:~$ docker container inspect rep-svr | grep IPAddress

            "SecondaryIPAddresses": null,
            "IPAddress": "",
                    "IPAddress": "172.18.0.5",

user@ubuntu:~$
```

Now use `curl` to retrieve one of the trash can reports (use an ID that you see from one of the logs):

```
user@ubuntu:~$ curl -s http://172.18.0.5:9090/reports/81

{"can_id":"81", "level":"52.01598882368821"}

user@ubuntu:~$
```

It works!

Now try an ID that doesn't exist:

```
user@ubuntu:~$ curl -s http://172.18.0.5:9090/reports/0

Error: Trash Can not found

user@ubuntu:~$
```

Perfect!

We can now kill all of our containers and then restart them without losing data because the data is persisted in a volume. In our simple case the volume is just stored on the host.

You can use *volume inspect* to locate the data:

```
user@ubuntu:~$ docker volume inspect report-data

[
    {
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/report-data/_data",
        "Name": "report-data",
        "Options": {},
        "Scope": "local"
    }
]

user@ubuntu:~$
```

List the files in the volume directory to see the Redis backing store:

```
user@ubuntu:~$ sudo ls -l /var/lib/docker/volumes/report-data/_data

total 4
-rw-r--r-- 1 999 docker 2324 Mar 22 04:04 appendonly.aof

user@ubuntu:~$
```

4. Cleanup

We can stop all of our containers at once by feeding the running container IDs to the docker stop command.

Try it:

```
user@ubuntu:~$ docker container stop $(docker container ls -q)

90d7e2c9bea9
9f1496b4e8a0
5eb903eacb10
44a6070943a8
```

```
user@ubuntu:~$
```

Docker will send SIGINT to each container and then give it 10 seconds to stop. If a container does not terminate within the 10 seconds Docker sends SIGKILL, terminating the container immediately.

Now check in your new stateful report service source code:

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

On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   rep.js

no changes added to commit (use "git add" and/or "git commit -a")

user@ubuntu:~/trash-can/report$ git commit -am "adding persistant storage"

[master aafe3fd] adding persistant storage
 1 file changed, 28 insertions(+), 7 deletions(-)
 rewrite rep.js (97%)
```

Congratulation you have completed Lab 6!

[OPTIONAL] Test the Independence of Containers And Volumes

Use the `docker container rm` command to delete your report server and your redis server. Then run the respective images to create new containers, attaching the redis container to the original volume. Now retrieve some old data (saved before they the new containers were created) to demonstrate the stateful persistence of your solution.