RX-M Cloud Native Consulting

Microservices

Lab 9 – API Gateways

An API gateway acts as an entry point into a context for clients outside of that context. For example, a mobile phone based client app may want to reach a microservices based application running in a cloud hosted cluster. External clients, such as the mobile phone, may not have the ability to discover or route traffic to the vast array of microservices composing an application, or may not want to, due to the complexity of the task. An API gateway can act as a single point of ingress for all traffic entering the application context, simplifying the mobile app access to the application and insulating the mobile app from changes within the microservice fabric.

API gateways may handle requests in various ways:

- Requests may be proxied/routed to an appropriate service
- Requests may be fanned out to multiple services
- Requests may be translated, invoking different backend services based on input
- Requests may be filtered based on security considerations
- etc.

Nginx is a powerful web server and can also act as a basic API gateway. In this lab we will create two services and then configure Nginx as an API gateway to front-end our services for outside clients.

1. Create Service A

To begin create a simple Node.js service that returns an identifying string:

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

user@ubuntu:~/trash-can$ mkdir svra

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

```
user@ubuntu:~/trash-can/svra$ vim svra.js
user@ubuntu:~/trash-can/svra$ cat svra.js
var express = require('express');
var http = require('http');

var app = express();

app.get('/status', function(req, res) {
    return res.send('Server A\n');
});

http.createServer(app).listen(9090, function() {
    console.log('Listening on port 9090');
});
user@ubuntu:~/trash-can/svra$
```

Our simple service listens on port 9090 and responds to requests against the status IRI with the string "Server A".

Next, package the service in a container:

```
user@ubuntu:~/trash-can/svra$ vim Dockerfile
user@ubuntu:~/trash-can/svra$ cat Dockerfile
FROM node
RUN npm install express
COPY . /
CMD node svra.js
user@ubuntu:~/trash-can/svra$
```

```
user@ubuntu:~/trash-can/svra$ docker image build -t svr:a .
Sending build context to Docker daemon 3.072 kB
Step 1/4 : FROM node
 ---> 47522eb1edb5
...
Step 4/4 : CMD node svra.js
 ---> Running in 47c08f07c26e
 ---> f35608c7d5fd
Removing intermediate container 47c08f07c26e
Successfully built f35608c7d5fd
user@ubuntu:~/trash-can/svra$
```

© Copyright Thursday, Feb 14, 2019, 7:00 AM RX-M LLC

Finally run and test your new service:

```
user@ubuntu:~/trash-can/svra$ docker container run -d --name=svra1 svr:a
c4d951aa38680c44f3765ce0777e9e1794789072501a890eb77723f16d1eac25

user@ubuntu:~/trash-can/svra$ docker container inspect svra1 | grep -i IPAddress
            "SecondaryIPAddresses": null,
            "IPAddress": "172.17.0.2",
                    "IPAddress": "172.17.0.2",

user@ubuntu:~/trash-can/svra$ curl http://172.17.0.2:9090/status
Server A
```

Great, step one complete!

2. Create Service B

Now create a second service just like the first that returns the string "Server B":

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

user@ubuntu:~/trash-can$ mkdir svrb

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

```
user@ubuntu:~/trash-can/svrb$ vim svrb.js
user@ubuntu:~/trash-can/svrb$ cat svrb.js
var express = require('express');
var http = require('http');

var app = express();

app.get('/status', function(req, res) {
    return res.send('Server B\n');
});

http.createServer(app).listen(9090, function() {
    console.log('Listening on port 9090');
});
user@ubuntu:~/trash-can/svrb$
```

Now create the image.

```
user@ubuntu:~/trash-can/svrb$ vim Dockerfile

user@ubuntu:~/trash-can/svrb$ cat Dockerfile
FROM node
RUN npm install express
COPY . /
CMD node svrb.js

user@ubuntu:~/trash-can/svrb$ docker image build -t svr:b .
Sending build context to Docker daemon 3.072 kB
Step 1/4 : FROM node
 ---> 47522eb1edb5
Step 2/4 : RUN npm install express
 ---> Using cache
 ---> ed1f152475a4
Step 3/4 : COPY . /
 ---> 1c2979612ad3
Removing intermediate container f4d6d513d177
Step 4/4 : CMD node svrb.js
 ---> Running in 25cc87ddd98b
 ---> ae8a0ebbf453
Removing intermediate container 25cc87ddd98b
Successfully built ae8a0ebbf453

user@ubuntu:~/trash-can/svrb$ docker container run -d --name=svrb1 svr:b
c519a08e4f33c25d1bb366baafb5003cd5d85eadb31b363974158029b47fc560

user@ubuntu:~/trash-can/svrb$ docker container inspect svrb1 | grep -i IPAddress
            "SecondaryIPAddresses": null,
            "IPAddress": "172.17.0.3",
                    "IPAddress": "172.17.0.3",
```

© Copyright Thursday, Feb 14, 2019, 7:00 AM RX-M LLC

```
user@ubuntu:~/trash-can/svrb$ curl http://172.17.0.3:9090/status
Server B
```

Perfect!

3. Create an API Gateway

The last step is to create an API gateway that clients can connect to as a proxy for our entire application. Our API gateway will allow end users to reach both of our back-end services using different IRIs.

- `/trashlevel` will be used to forward users to the A service
- `/reports` will be used to forward users to the B service.

We can use Nginx as a proxy pass through server to perform our simple API gateway task. We can pull an Nginx image from Docker Hub but we will need to create a configuration file that tells Nginx how we want to forward traffic. Create the following Nginx configuration file:

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

user@ubuntu:~/trash-can$ mkdir apigw

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

user@ubuntu:~/trash-can/apigw$ vim default.conf
user@ubuntu:~/trash-can/apigw$ cat default.conf
server {
    location /trashlevel {
        rewrite ^/trashlevel(.*) $1 break;
        proxy_pass http://172.17.0.2:9090;
    }

    location /reports {
        rewrite ^/reports(.*) $1 break;
        proxy_pass http://172.17.0.3:9090;
    }
}
user@ubuntu:~/trash-can/apigw$
```

This `default.conf` configuration file can be copied into `/etc/nginx/conf.d` (in our image) to tell Nginx to perform our two forwarding requests. The first location stanza takes traffic hitting the `/trashlevel` IRI and then rewrites the IRI, stripping the "/trashlevel" string out. So if we curl "/trashlevel/status", nginx will forward to http://172.17.0.2:9090/status.

Now create an Nginx container that uses our new configuration:

```
user@ubuntu:~/trash-can/apigw$ vim Dockerfile
user@ubuntu:~/trash-can/apigw$ cat Dockerfile
FROM nginx
COPY ./default.conf /etc/nginx/conf.d/

user@ubuntu:~/trash-can/apigw$ docker image build -t apigw:v1.0 .
Sending build context to Docker daemon 3.072 kB
Step 1/2 : FROM nginx
latest: Pulling from library/nginx
Digest: sha256:52a189e49c0c797cfc5cbfe578c68c225d160fb13a42954144b29af3fe4fe335
Status: Downloaded newer image for nginx:latest
 ---> 6b914bbcb89e
Step 2/2 : COPY ./default.conf /etc/nginx/conf.d/
 ---> a1add1ad4547
Removing intermediate container ee93a3aff618
Successfully built a1add1ad4547
user@ubuntu:~/trash-can/apigw$
```

Now run your gateway container and test it:

```
user@ubuntu:~/trash-can/apigw$ docker container run -d -p 80:80 apigw:v1.0
4afdd9d63ecdfe0481def5cdb967410891121037b8775b28c9a79dd4b00a2926

user@ubuntu:~/trash-can/apigw$ curl http://localhost/trashlevel/status
Server A

user@ubuntu:~/trash-can/apigw$ curl http://localhost/reports/status
Server B
```

We now have a single end point that proxies client traffic between our trash level and reports services!

© Copyright Thursday, Feb 14, 2019, 7:00 AM RX-M LLC

Congratulations, you have completed the API Gateway lab!

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