Using Consul service discovery with Docker on AWS

Hagai Kahana Follow
Jan 14 · 16 min read

In the last post, we explored how to run multiple containers on a single host with an application running in one container that persists data on MongoDB running on another container.

In this post, we will launch our system in a cloud environment and have a dedicated instance for each docker container. This will represent a deployment that is closer to a production environment. We will use AWS as this is currently the largest cloud provider and the one we are most familiar with.

Note: There are many options for running and orchestrating Docker containers. Some are well-known configuration management and orchestration tools such as Chef, Puppet, Salt and Ansible others are dedicated management of containerized applications such as Kubernetes, Docker swarm or managed platform such as AWS ECS. We will not use those but rather launch them manually ourselves. This will not be the recommended approach when running a production environment, but this is just a learning exercise for us.

Our system is composed of a Node.JS application in the frontend talking with a MongoDB storage repository. One interesting problem to

resolve in a distributed application is how to orchestrate the environment in a manner that one service can find another service that it depends on.

In our little example that would be reflected in how the Node.JS application can discover the MongoDB storage? Service Discovery, which is the mean an entity's ability to discover a service automatically, is a problem that can be resolved in many ways and different tools.

One option is to put one service behind a load balancer (Nginx, F5 BIG-IP, ELB) which will publish a VIP and have dependent components pass communicate via the VIP to the service hosts. Another option is using tools such as Hashicorp Consul, CoreOS Etcd, Apache Zookeeper or Netflix Eureka.

There are many articles comparing those tools and for this post, we will demonstrate the use of Consul due to it's integrated support with DNS that allow a seamless discovery experience.

Consul features

Consul has three main features:

1. **Service discovery**—Allow services to register and clients to use DNS or HTTP to discover registered services.
2. **Health Checking**—Ability to define health check cooperation that consul can execute and define the health state of a monitored node. Once a node is considered unhealthily consul can stop

routing traffic to that node. (The node address will not be published during service discovery request by other clients)

3. **KV Store**—Client can store and publish Key/value data, this can be useful for configuration feature flag and other means of communication between nodes.

We will only focus on the first feature the Service discovery in this example. But once Consul is setup it will be easy to utilize the health check KV store and any other capability provided by this system.

Consul agent types

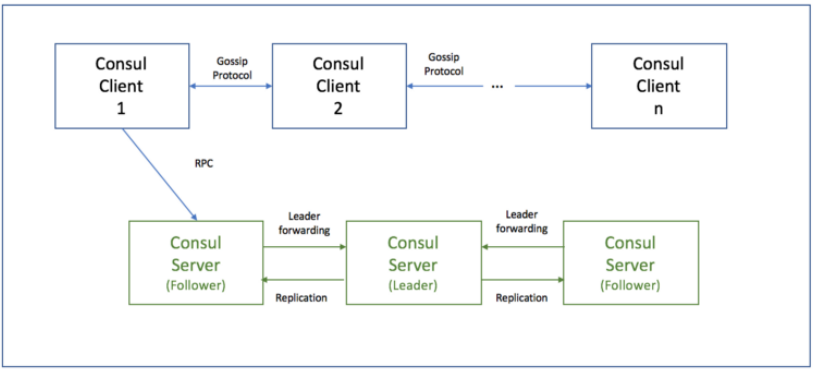
There are fundamental concepts that we need to understand about Consul before we can effectively use Consul.

Consul is running as a background daemon on a running host, a Consul agent can run in two modes, client or server.

Client agent—Accept a request and forwards them to a server agent for a reply. Usually, an application communicates with the locally running Consul agent client to discover other services by using HTTP or DNS requests (or using K/V storage). There is no penalty for running a large fleet of consul clients and they can scale along with your service hosts.

Server agent—Communicates with other server agents and creates a Raft peer set. The server agents will elect one leader out of the set. They are also responsible to reply to RPC queries. It is recommended to have between 3–5 server agents. This will ensure good performance for

reaching consensus between server nodes, as there is still a small number of servers. But, it will still protect the consul cluster against single server failure.

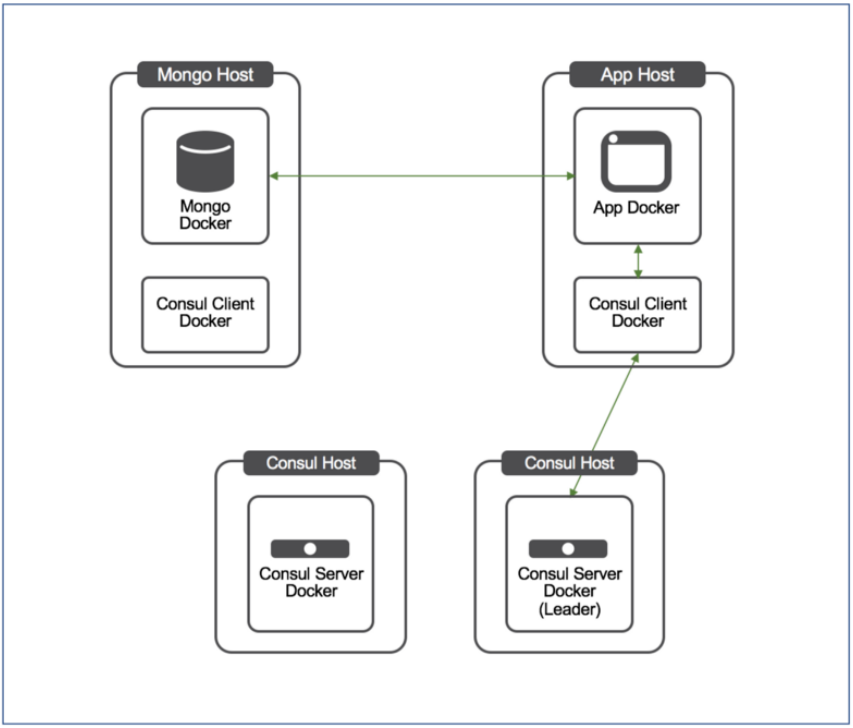


Now that we are familiar with Consul let's go ahead and use in our system. We would like to use Consul service discovery feature to help the application nodes find the repository nodes.

Objective

Demonstrate service discovery using Consul with dockers on AWS. Launching an express Node.JS application that discovers a MongoDB service to persists information.

System diagram



Each EC2 Instance will run one or more docker container

1. Consul Host 1—Runs a leader Consul server docker container
2. Consul Host 2—Runs a follower Consul server docker container
3. Mongo Host—Runs a consul client docker and a Mongo docker that publish itself as a **db**.service.consul using the local consul client docker.
4. App Host—Runs a Consul client docker and an application docker containing our login page. The application docker will query the

local consul client via DNS query the location of **db**.service.consul.

Note: In production, you would run 3 or 5 consul servers but for simplicity, in this example, we are only running 2. Also, this model ensures that we can horizontally scale the Application Host, but you cannot scale the Mongo Host without using replication or sharding, which is out of the scope for this post.

Prerequisites

- GIT
- AWS account

Warning**:** Running this system with EC2 instances on AWS will subject to a small cost. There is the option to use free tier instance type if you are eligible but if you are not I would recommend using the cheapest "nano.t2" type.

Setup consul servers

First, we build our Consul servers. This must be orchestrated before running the application or MongoDB hosts because they are running Consul clients that need to connect to a Consul server to operate correctly.

Setup First Consul Server

Login to your AWS account, choose a region and go the "EC2" service. From the Amazon EC2 console dashboard, choose **Launch Instance**.

Step 1: Choose an Amazon Machine Image (AMI)

We will choose an **Amazon Linux AMI**, that is the top item on that list, the AMI id will be different per region.

Step 2: Choose an Instance Type

Choose an instance type, we choose “t2.nano” due to cost but if you have ‘free-tier’ hours you might want to choose ‘free-tier’ eligible type “t2.micro”.

Step 3: Configure Instance Details

Open the “Advanced Details”, one of the fields is User data text box. The User data allows us to provide a script to setup the EC2 instance during its first boot.

Use that field to configure our Consul server with the following bash script:

User data bash script for consul server

```
#!/bin/bash
exec > >(tee /var/log/user-data.log|logger -t user-data -s
2>/dev/console) 2>&1
```

```
echo "Install and start docker"
yum update -y
yum install -y docker
service docker start
usermod -a -G docker ec2-user
```

```
echo "Get host Public IP"
export EC2_PUBLIC_IP="$(curl
http://169.254.169.254/latest/meta-data/public-ipv4)"
```

```
echo "Run Consul server"
docker run -d -h $HOSTNAME -p 8300:8300 -p 8301:8301 -p
8301:8301/udp -p 8302:8302 -p 8302:8302/udp -p 8400:8400 -p
8500:8500 -p 53:8600/udp --name consul-srv
gliderlabs/consul-server -server -advertise $EC2_PUBLIC_IP -
bootstrap-expect 2
```

Script details

```
exec > >(tee /var/log/user-data.log|logger -t user-data -s
2>/dev/console) 2>&1
```

Logs the script output and error to /var/log/user-data.log file

```
echo "Install and start docker"
yum update -y
yum install -y docker
service docker start
usermod -a -G docker ec2-user
```

Using yum, updates libraries, install and run dockers which the libraries and add ec2-user to the docker group.

```
echo "Get host Public IP"
export EC2_PUBLIC_IP="$(curl
http://169.254.169.254/latest/meta-data/public-ipv4)"
```

Getting instance public IP address using EC2 instance metadata

```
echo "Run Consul server"
docker run -d -h $HOSTNAME -p 8300:8300 -p 8301:8301 -p
8301:8301/udp -p 8302:8302 -p 8302:8302/udp -p 8400:8400 -p
8500:8500 -p 53:8600/udp --name consul-srv
gliderlabs/consul-server -server -advertise $EC2_PUBLIC_IP -
bootstrap-expect 2
```

Using ‘docker run’ command to run our Consul server. We are using gliderlabs/consul-server docker image.

docker run parameters:

-d—runs in detach mode, the containers will exit when the consul-server process running inside the container exits.

-h—defines the container hostname

-p—map multiple ports to the container.

consul server parameter:

-server—Specify that this is a Consul server agent, not a client

-advertise—Specify the IP address we advertise to other Consul nodes in the cluster

-bootstrap-expect—The number of expected Consul servers peer set.

Step 4: Add Storage

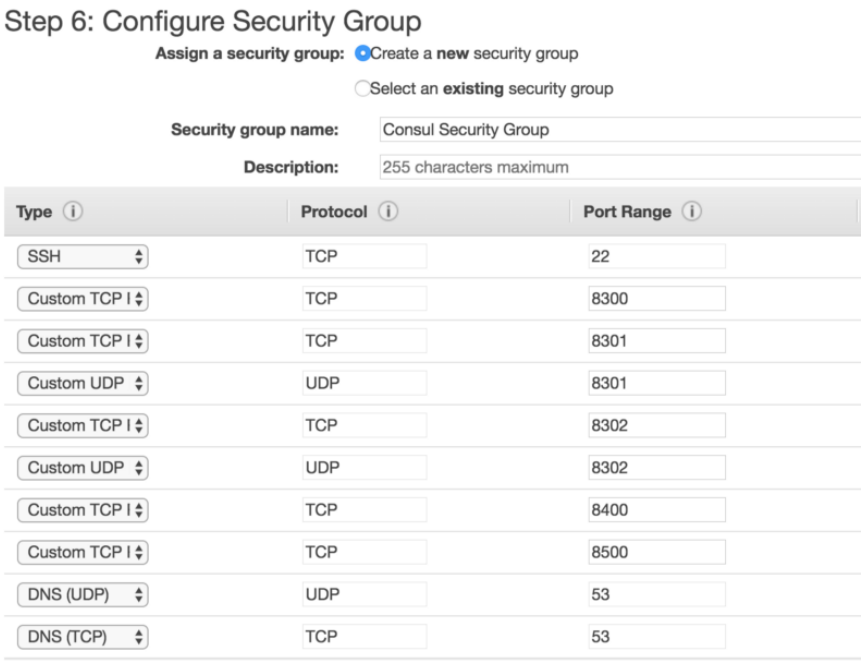
Use default values

Step 5: Add Tags

Add a tag with key “Name” and value of “Consul Server main”. This will allow us to easily identify the Consul server instance in the grid view.

Step 6: Configure Security Group

As we saw in the “docker run” command the Consul server needs several ports to be opened.



Details on the user of the ports can be found here.

Step 7: Review Instance Launch

Make sure to review and use a key pair you have access to or download a new one.

Validation

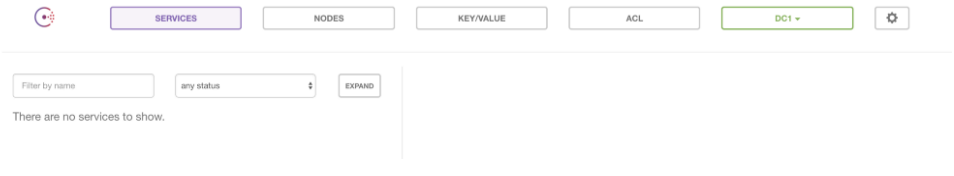
Let's ensure that the Consul server is running correctly by login to the newly created instance and listing the running docker containers on the instance. We should see our gliderlabs/consul-server named consul-srv as the only running container on that list.

```
bash-3.2$ ssh -i ~/aws/mac-key-pair.pem  ec2-user@54.215.252.196
Last login: Sun Dec 24 22:50:19 2017 from 50.206.82.181

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2017.09-release-notes/
[ec2-user@ip-172-31-22-20 ~]$ docker ps
CONTAINER ID        IMAGE                     COMMAND                  CREATED             STATUS              PORTS
                                                                NAMES
37cd2c70919a        gliderlabs/consul-server   "/bin/consul agent..."   5 minutes ago       Up 5 minutes        0.0.0.0:8300-8302->8300-8302/tcp, 0.0.0.0:8400->8400/tcp, 0.0.0.0:8500->8500/tcp, 0.0.0.0:
8301-8302->8301-8302/udp, 8600/tcp, 0.0.0.0:53->8600/udp   consul-srv
```

If there are issues we would check the logs at “/var/log/user-data.log” for details on the executing of the user data boot script.

Once we saw that the docker image is running we will try to us the Web UI by browsing the instance IP on port 8500.

Find the the newly launched EC2 instance public IP address and login using port 8500, *http://<IP-adress>:8500*.



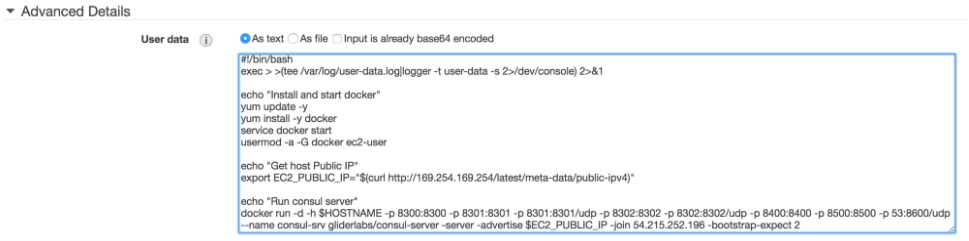
Web UI services list of Consul server

Web UI nodes list of Consul server

In both the Services and Nodes tab, there are no entries as the Consul server peer set is not complete, we expect two servers to be up and running (base on the **-bootstrap-expect** value) but only have one. In order to complete the peer set, we will need to launch another Consul server.

Setup Second Consul Server

We are ready to launch our second Consul server to ensure the cluster is ready to serve Consul clients. We specify only steps that differ from the first Consul service launch, the other steps in the AWS launch wizards that are not specified should have the same configuration as the first Consul server.
The User data script used to launch this Consul server is this:




```
#!/bin/bash
exec > >(tee /var/log/user-data.log|logger -t user-data -s
2>/dev/console) 2>&1
```

```
echo "Install and start docker"
yum update -y
yum install -y docker
service docker start
usermod -a -G docker ec2-user
```

```
echo "Get host Public IP"
export EC2_PUBLIC_IP="$(curl
http://169.254.169.254/latest/meta-data/public-ipv4)"
```

```
echo "Run consul server"
docker run -d -h $HOSTNAME -p 8300:8300 -p 8301:8301 -p
8301:8301/udp -p 8302:8302 -p 8302:8302/udp -p 8400:8400 -p
8500:8500 -p 53:8600/udp --name consul-srv
gliderlabs/consul-server -server -advertise $EC2_PUBLIC_IP -
join 54.215.252.196 -bootstrap-expect 2
```

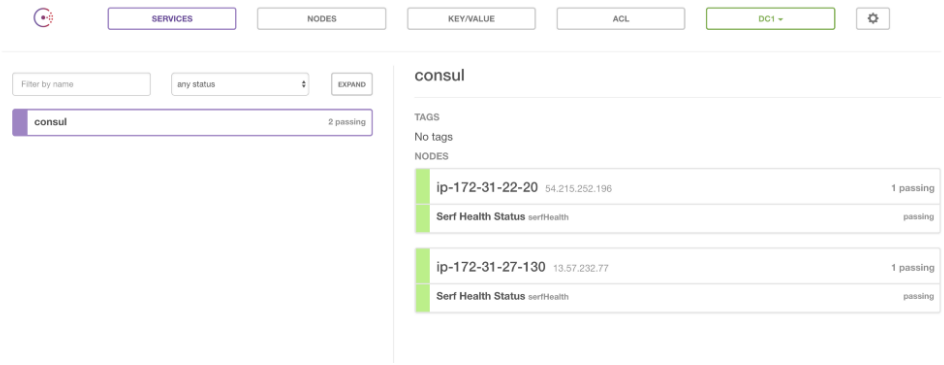
Script details


```
echo "Run consul server"
docker run -d -h $HOSTNAME -p 8300:8300 -p 8301:8301 -p
8301:8301/udp -p 8302:8302 -p 8302:8302/udp -p 8400:8400 -p
8500:8500 -p 53:8600/udp --name consul-srv
gliderlabs/consul-server -server -advertise $EC2_PUBLIC_IP -
join 54.215.252.196 -bootstrap-expect 2
```


The only change to the script is the “Run Consul Server” section. We added a flag **-join** and provide the address of the Consul server agent to join when booting.

In AWS launch wizard, add a tag with key “Name” and value of “Consul Server secondary”. This will allow us to easily identify the instance in the grid view.

After the server instance is running we can check the Web UI again and expect to see the “consul” service running



Web UI services list of Consul server

It includes the two instances running and contain Serf Health Status details.

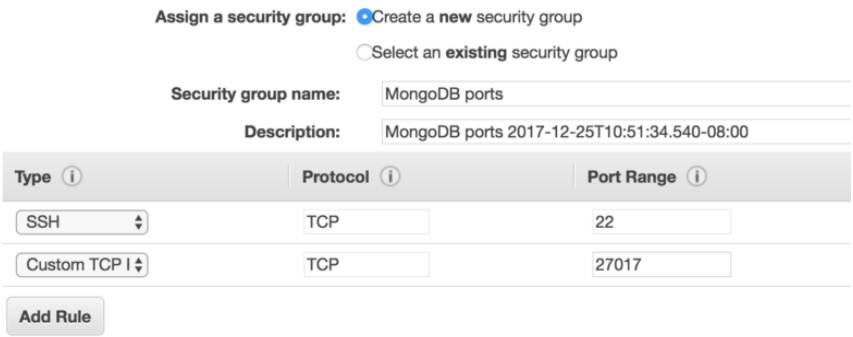
Now, we have a Consul that our instances can use to register services and discover them.

Setup MongoDB service

We will start by launching a MongoDB instance and register it as a DB service. That means that anyone that wants to find about the MongoDB service can get its IP address by query DNS for the following domain **db**.service.consul.

In the AWS launch wizards tag “Name” should be descriptive as in “MongoDB”

The security group should open the default MongoDB port



The User data script attached to the instance is:

```
#!/bin/bash
exec > >(tee /var/log/user-data.log|logger -t user-data -s
2>/dev/console) 2>&1
```

```
echo "Install and start docker"
yum update -y
yum install -y docker
```

```
service docker start
usermod -a -G docker ec2-user
```

```
echo "Get host Public IP"
export EC2_PUBLIC_IP="$(curl
http://169.254.169.254/latest/meta-data/public-ipv4)"
```

```
echo "Run mongo docker"
docker volume create --name mongo_storage
docker run --name mongo --hostname mongoNode -v
mongo_storage:/data -p 27017:27017 -d mongo --smallfiles --
storageEngine wiredTiger --port 27017 --bind_ip 0.0.0.0
```

```
#Change folder owner to the user container
docker exec mongoNode bash -c 'chown -R mongodb:mongodb
/data'
```

```
echo "Create user for DB passport"
sleep 2
docker exec mongo bash -c "mongo passport --eval
'db.createUser({ user: \"admin\", pwd: \"abc123\", roles: [
{ role: \"readWrite\", db: \"passport\" } ] });'"
```

```
echo "Run consul agent docker"
docker run -d -h $HOSTNAME -p 8300:8300 -p 8301:8301 -p
8301:8301/udp -p 8302:8302 -p 8302:8302/udp  -p 8500:8500 -p
53:8600/udp --name consul-agent consul agent -advertise
$EC2_PUBLIC_IP -join 54.215.252.196 -client 0.0.0.0
```

```
echo "Register mongo as a service in consul"
curl -X PUT http://localhost:8500/v1/agent/service/register
-H "Content-Type: application/json" --data '{"ID":
"db1","Name": "db","Port": 27017}'
```

Script details

```
docker volume create --name mongo_storage
```

Create a docker volume, this will allow us to map to the /data MongoDB folder and ensure that the database persists beyond the lifetime docker container lifetime.

```
docker run --name mongo --hostname mongoNode -v
mongo_storage:/data -p 27017:27017 -d mongo --smallfiles --
storageEngine wiredTiger --port 27017 --bind_ip 0.0.0.0
```

Launch the MongoDB docker with the volume mapped to /data folder, port 27017 mapped to the docker container and bind_ip to all network interfaces that are available. (This will be publicly exposed to all IPs on the internet and has security implication, especially for a production environment)

```
docker exec mongoNode bash -c 'chown -R mongodb:mongodb
/data'
```

Modify the data folder ownership to the MongoDB user, which is the container user. This is required as the MongoDB user will be the one that will be accessing the folder and its files.

```
echo "Create user for DB passport"
sleep 2
docker exec mongo bash -c "mongo passport --eval
'db.createUser({ user: \"admin\", pwd: \"abc123\", roles: [
{ role: \"readWrite\", db: \"passport\" } ] });'"
```

This section creates a user for the passport DB. It waits for 2 seconds to allow the time for Docker container to load. After the sleep, it executes a bash command that uses mongo shell to log to passport DB and create a user with read-write permissions.

```
echo "Run consul agent docker"
docker run -d -h $HOSTNAME -p 8300:8300 -p 8301:8301 -p
8301:8301/udp -p 8302:8302 -p 8302:8302/udp  -p 8500:8500 -p
53:8600/udp --name consul-agent consul agent -advertise
$EC2_PUBLIC_IP -join 54.215.252.196 -client 0.0.0.0
```

Start our Consul client agent. We are using ‘consul’ docker image with the same open ports as Consul server. The Consul parameters advertise the instance public IP address and join the first Consul server that we created. Also, we do not have the **-server** flag which indicates that this is a client Consul agent.

```
echo "Register mongo as a service in consul"
curl -X PUT http://localhost:8500/v1/agent/service/register
-H "Content-Type: application/json" --data '{"ID":
"db1","Name": "db","Port": 27017}'
```

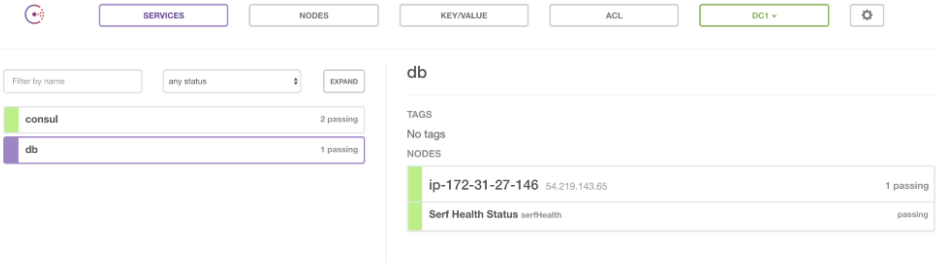
Finally, we are registering a DB service on port 27017, the port that is mapped to the MongoDB docker container.

Validation

After the instance has completed it's launch make sure we can connect to it. Assuming that we have a local installation of MongoDB we can use the shell to connect to the remote MongoDB host by running this command

```
mongo 54.219.143.65:27017/passport -u admin -p abc123
```

And ensure we can login in a similar manner that the application will connect to the db service.
Examine the Consul WebUI again to validate that the db service was registered correctly



Web UI services list of Consul server

We can see in the services tab that we have another service call “db” with a single host that matches the Public IP address of our MongoDB host.
Finally, we can also on a local machine resolve the “db.service.consul” domain name using one of the Consul servers as the DNS server

```
bash-3.2$ dig db.service.consul @54.215.252.196

; <<>> DiG 9.8.3-P1 <<>> db.service.consul @54.215.252.196
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 17027
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;db.service.consul.             IN      A

;; ANSWER SECTION:
db.service.consul.      0       IN      A       54.219.143.65

;; Query time: 36 msec
;; SERVER: 54.215.252.196#53(54.215.252.196)
;; WHEN: Mon Dec 25 11:07:41 2017
;; MSG SIZE  rcvd: 68
```

We get in the ANSWER SECTION reply the IP address of our MongoDB. In that case, our application could use DNS query for the MongoDB service discovery.

Setup Application host

Next, we will launch our application instance which is the same login application example from the previous post. Use the following command to download the code:

```
git clone https://github.com/hagaik/easy-node-
authentication.git
```

Once we have the code we need to define the MongoDB URL. Go to file 'config/database.js' and modify it to contain the following configuration:

```
// config/database.js
module.exports = {
    'url' :
'mongodb://admin:abc123@db.service.consul:27017/passport'
};
```

The application will locate the MongoDB service by searching for the "db.service.consul" domain name. We get access to the passport DB by

using the user admin that was created during MongoDB instance boot time.

We can commit the code and publish the node package and have the EC2 application host use npm to download and run the application. But we would rather have the application “Dockerize” and run it in next to a Consul client in a similar manner that we ran MongoDB docker alongside the Consul client.

To achieve that we create a Docker file

```
FROM node:carbon

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
COPY package*.json ./

RUN npm install

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "npm", "start" ]
```

Hopefully, it is self-explanatory. We are installing the package and running npm install before copying our application code. This helps us

take advantage of Docker caching layers. Each command in the Docker file results in a different docker layer. We should expect that code to change frequently but not the package JSON files. This helps reduce the Docker build time as we will not run npm install every code change. See this article for more details

From there you can publish the image

```
$ docker push <YOU_DOCKER_ID>/auth-app-consul
```

If you do not want to create a dedicate image feel free to use the uploaded docker image.

How do we ensure that our application can communicate with the local Consul client container to retrieve the MongoDB address?

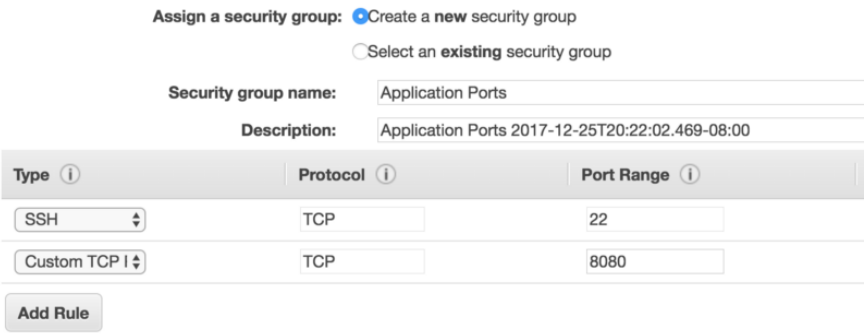
For that, we will follow the excellent guide defined in this post.

1. Create a dummy IP where the Consul client agent binds the client interfaces such as DNS.
2. Launch our docker application to use the dummy IP address to resolve DNS queries
3. Use dnsmasq to direct .consul domain queries to the Consul client agent

Now that we have a plan and our docker image with our application is ready we can run our application on a dedicated EC2 instance.

In the AWS launch wizards input run the same step as a Consul server. The tag “Name” should be descriptive as in “Application”

The security group should expose the TCP port 8080



The User data script attached to the instance is

```
#!/bin/bash
```

```
exec > >(tee /var/log/user-data.log|logger -t user-data -s
2>/dev/console) 2>&1
```

```
echo "Install and start docker"
yum update -y
yum install -y docker
service docker start
usermod -a -G docker ec2-user
```

```
echo "Get host Public IP"
export EC2_PUBLIC_IP="$(curl
http://169.254.169.254/latest/meta-data/public-ipv4)"
```

```
echo "create dummy interface"
ip link add dummy0 type dummy
ip addr add 169.254.1.1/32 dev dummy0
ip link set dev dummy0 up
```

```
echo "Install dnsmasq"
yum install -y dnsmasq
cat <<EOT>> /etc/dnsmasq.d/consul.conf
server=/consul/169.254.1.1#8600
listen-address=127.0.0.1
listen-address=169.254.1.1
EOT
service dnsmasq start
```

```
echo "Run consul docker"
docker run -d -h $HOSTNAME -p 8300:8300 -p 8301:8301 -p
8301:8301/udp -p 8302:8302 -p 8302:8302/udp  -p 8500:8500 -p
8600:8600/udp --name app-agent --net=host consul agent -
advertise $EC2_PUBLIC_IP -join 54.215.252.196 -client
169.254.1.1
```

```
echo "Run application docker"
docker run --dns=169.254.1.1 --name appNode -p 8080:8080 -d
hagaik/auth-app-consul
```

Script details

```
echo "create dummy interface"
ip link add dummy0 type dummy
```

```
ip addr add 169.254.1.1/32 dev dummy0
ip link set dev dummy0 up
```

Create a dummy IP address “169.254.1.1” on the host.

```
echo "Install dnsmasq"
yum install -y dnsmasq
cat <<EOT>> /etc/dnsmasq.d/consul.conf
server=/consul/169.254.1.1#8600
listen-address=127.0.0.1
listen-address=169.254.1.1
EOT
service dnsmasq start
```

Install, configure the DNS queries for “.consul” domain to be directed to the dummy IP and start the dnsmasq service.

```
echo "Run consul docker"
docker run -d -h $HOSTNAME -p 8300:8300 -p 8301:8301 -p
8301:8301/udp -p 8302:8302 -p 8302:8302/udp  -p 8500:8500 -p
8600:8600/udp --name app-agent --net=host consul agent -
advertise $EC2_PUBLIC_IP -join 54.215.252.196 -client
169.254.1.1
```

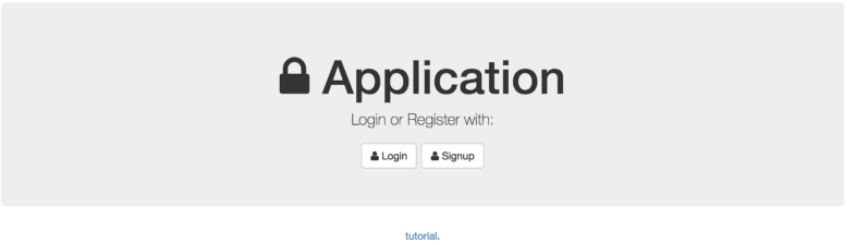
The Consul client agent is similar to the MongoDB Consul client agent with two changes. Its network is set to host and the client interface is bound to the dummy IP.

```
echo "Run application docker"
docker run --dns=169.254.1.1 --name appNode -p 8080:8080 -d
hagaik/auth-app-consul
```

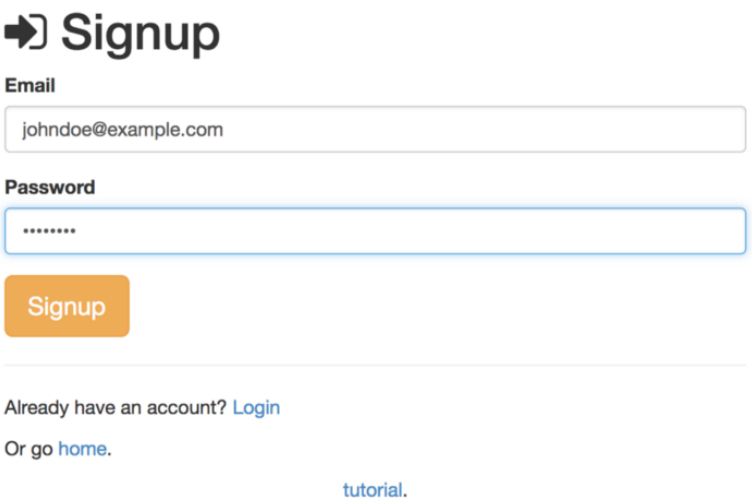
We are running our application using the uploaded docker image we created earlier and use the dummy ip as the DNS server.

Validation

Once the instances are done launching on AWS use the public IP to reach the application on port 8080.
You should be able to see the application running



From there we can Signup and create an account:



What should have happened is that the application used a DNS query that was directed to the Consul agent running on the host. The Consul client agent would talk to the Consul server for a reply and provide the resolution address to the application. From there the application has discovered the MongoDB host and can send the new user details.
Let's validate that by examining the data on the MongoDB EC2 instance and ensure that the new user was created.

```
$ mongo 54.219.143.65:27017/passport -u admin -p abc123
MongoDB shell version v3.4.10
connecting to: mongodb://54.219.143.65:27017/passport
MongoDB server version: 3.4.10
Welcome to the MongoDB shell.
> show dbs
admin     0.000GB
local     0.000GB
```

```
passport  0.000GB
> use passport
switched to db passport
> show collections
users
> db.users.find({})
{ "_id" : ObjectId("5a1f982615336e0010c38dd1"), "local" : {
"password" :
"$2a$08$NQ.b1vOliIXWGj1xtfnMiOosZ.DDoihB7Mm2KcngnHVud7rKQBaa
.", "email" : "johndoe@example.com" }, "__v" : 0 }
```

We use `mongo` shell command to log in to the MongoDB host.

Once we are logged we list the databases, switch to `passport` database and list all of the users to find the new record of johndoe@example.com.

Final comment

This was an exercise to test a service discovery using Consul a tool provided by HashiCorp. In it we created the system environment manually. This is not something that is recommended for a production environment. Some of the issue we would address if this needed to migrate to production:

1. We would use orchestration and automation tools to create the environment, for example using Terraform to launch the instances and Ansible to configure them.
2. There are several security issues that we did not attempt to address. For example, using an embedded password or source

address in the security group. This would not be recommended for production environment.

3. We used 2 Consul server, as we mentioned it is recommended to use 3 or 5 Servers in production.

***Note**: All instances, key pairs and other resources were deleted when this post was published and they are displayed only as an example.*

Thanks for reading this post, this was a fun exercise and I hope you found it interesting as well.