



CISCO

Cisco
Cloud Partner VT

Cloud Native
Development Lab

Overview.....	1
Step 1 - Docker Setup.....	6
Step 2 - Consul Setup.....	11
Step 3 - Fabio Setup.....	16
Step 4 - Redis Setup.....	20
Step 5 - Golang Setup.....	23
Step 6 - Bringing it all together.....	28
Step 7 - Where to now?.....	37
Step 8 - Other things to try.....	42

Overview

Lab Objective

The objective of this lab is to introduce some common tools, methodologies and design considerations when you start to build out a Cloud Native application. It will also allow you to become familiar with Docker, interacting with the services and an introduction to the Golang language.

What are the lab requirements

There are only two requirements for this lab, 1) is to have Docker installed and running. If configured correctly, you should be able to run native Docker commands at the terminal or command window and not need to use Sudo or Administrator for it to succeed. 2) is to have an active Internet connection to enable you to download some images from Docker Hub.

What components are we going to use

There are a large number of different tools we could use in this lab, for example Server Load Balancers and each one of them has many different flavours both Open Source and Commercial offerings. This lab has taken some of the most popular Open Source offerings at the time of writing to focus in on. You can however swap individual components out for alternatives, however this lab will not cover the integration points for other tools.



GitHub

GitHub is a web-based Git repository hosting service. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features.

<http://www.github.com>

Docker

Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries – anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment.

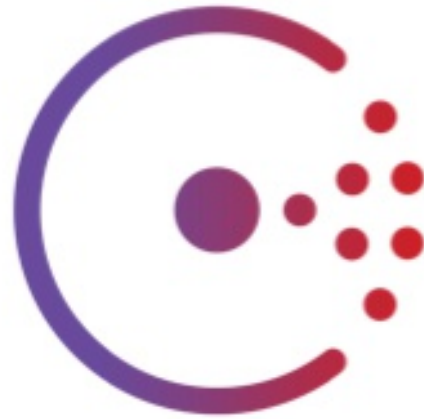
<http://www.docker.com>



Consul

Consul has multiple components, but as a whole, it is a tool for discovering and configuring services in your infrastructure. It provides several key features: Service Discovery, Health Checking, Key/Value Store, Multi Datacenter

<http://www.consul.io>



Fabio

fabio is a fast, modern, zero-conf load balancing HTTP(S) router for deploying applications managed by consul.

Register your services in consul, provide a health check and fabio will start routing traffic to them. No configuration required. Deployment, upgrading and refactoring has never been easier.

<https://github.com/eBay/fabio>



Redis

Redis is an open source (BSD licensed), in-memory data structure store, used as database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries.

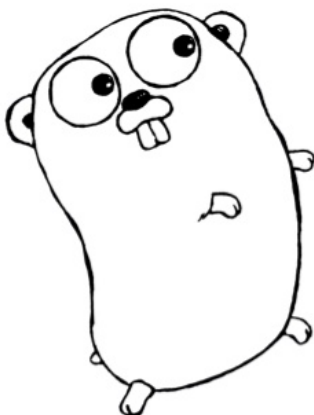
<http://www.redis.io>



Golang

Go, also known as golang, is a computer programming language whose development began in 2007 at Google, and it was introduced to the public in 2009. Go's three lead developers at Google were Robert Griesemer, Rob Pike, and Ken Thompson.

<http://golang.org>



What is the lab outcome?

At the end of this lab, you will have hopefully gained a good understanding of Docker and Docker parameters, how to setup supporting tools such as Consul and how automatic and manual registration occurs, how to create and compile GO code and insert into a container.

If you would like to see a graphical output of what you are going to produce, you can see a quick spoiler at the end of step 6 in this document.

Any assumptions about the lab?

The lab has been designed that if you have no or little experience of these components, you will be able to follow through step-by-step and get to the same output as someone who has good knowledge of all of these tools.

This lab has been written on and for a MAC. Other *nix platforms should be able to follow the commands exactly, however Windows users will need to amend the paths used.

Through this lab we will use and refer to certain syntax, boxes and commands, below is a summary these for your reference;

```
#
```

Any commands that you need to enter will be shown in boxes as shown above.

A single # will mean to be entered onto the host machine, <NAME># will refer to commands that need to be entered into the respective container, for example;

```
webserver#
```

Will mean the following command will need to be entered into the docker container called webserver.

Throughout this lab guide there will also be boxes with additional information and tips and some text that will be highlighted to signify its importance, they will look like the following;

Important information will be presented in boxes like this.

Important information will be in here

Step 1

Docker Setup



Docker

Step 1 - Docker Setup

At this stage you should have a working copy of docker installed either on your machine or accessible from your machine.

To check this is working as expected, please try the following;

```
# docker version
```

```
Client:
Version:      1.12.1
API version:  1.24
Go version:   gol.7.1
Git commit:   6f9534c
Built:        Thu Sep  8 10:31:18 2016
OS/Arch:      darwin/amd64


Server:
Version:      1.12.1
API version:  1.24
Go version:   gol.6.3
Git commit:   23cf638
Built:        Thu Aug 18 17:52:38 2016
OS/Arch:      linux/amd64
```

If you received some output similar to the above, then this lab is going to be very easy!

Please note you may receive different versions than the above, this is not too much of a concern as we will not be using any version specific features.

Step 1 - Docker Setup

Lets make sure we have a clean environment for running this lab, so make sure we have no running containers and no images available. (This is optional, but is going to make life easier for this lab)

docker ps -a  We will be using this command a lot.

The output from the command should look similar to the below;

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

Now check the images available;

docker images

The output from the command should look similar to the below;

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
------------	-----	----------	---------	------

If you need to remove a running container, copy the ID from the "CONTAINER ID" column in the first command and use it in the following format;

docker rm -f <ID>

If you need to remove an image, copy the ID from the "IMAGE ID" column in the second command and use it in the following format;

docker rmi <ID>

Finally rerun the previous two commands to ensure you have a clean environment;

docker ps -a
docker images

Step 1 - Docker Setup

We can now pull a couple of images we will need for this lab. Please be patient, it will take a few minutes for each of these commands to complete;

```
# docker pull golang
```

During this process you should see output that resembles;

```
> docker pull golang
Using default tag: latest
latest: Pulling from library/golang

8ad8b3f87b37: Already exists
751fe39c4d34: Already exists
ae3b77eefc06: Downloading [====>] 2.57 MB/42.5 MB
92c7f8737c98: Downloading [=====>] 14.06 MB/56.9 MB
bf37bbda794c: Downloading [==>] 2.153 MB/81.63 MB
6e9d2df2553b: Waiting
a79803310595: Waiting
```

Continue and pull the other four images we need;

```
# docker pull consul
```

And

```
# docker pull redis
```

And

```
# docker pull magiconair/fabio
```

And

```
# docker pull ubuntu
```

Step 1 - Docker Setup - Troubleshooting

You may receive an error which states something like;

"Cannot connect to the Docker daemon. Is the docker daemon running on this host?"

This is to do with the docker process not running or not being linked correctly. If running on a Mac, Check your output looks like the following;

```
# ls -la /var/run/docker.sock  
  
lrwxr-xr-x 1 root daemon 59 2 Aug 11:44 /var/run/docker.sock ->  
/var/root/Library/Containers/com.docker.docker/Data/s60
```

This is to do with a link being created incorrectly. The important part to check is the piece highlighted, if your output matches this, please proceed to fix it with the following;

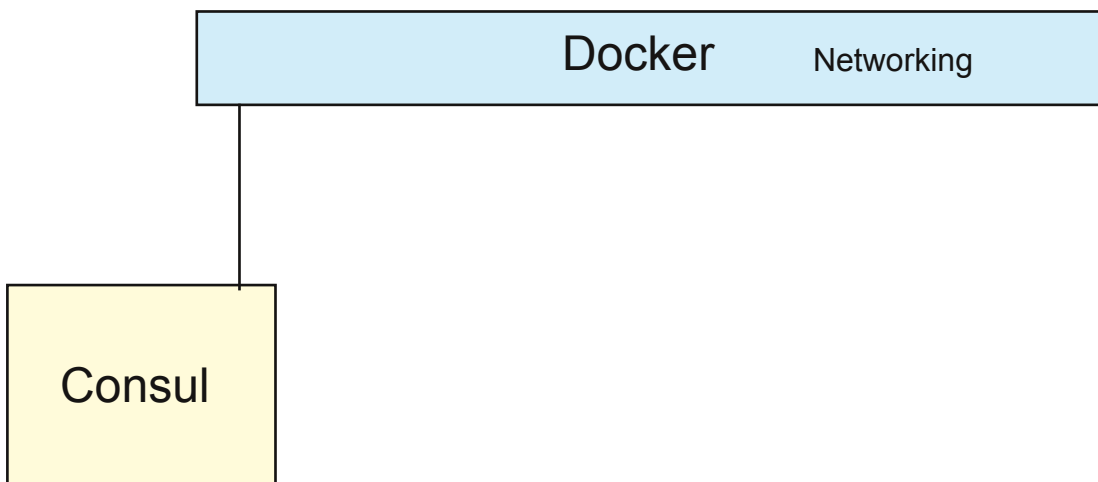
```
# sudo ln -sf ~/Library/Containers/com.docker.docker/Data/s60 /var/run/docker.sock
```

Afterwards the output should look like the following;

```
# ls -la /var/run/docker.sock  
  
lrwxr-xr-x 1 root daemon 59 2 Aug 11:44 /var/run/docker.sock ->  
/Users/<USERNAME>/Library/Containers/com.docker.docker/Data/s60
```

Step 2

Consul Setup



Step 2 - Consul Setup

Consul does not need any specific configuration or changes to be made for this lab, so we can go ahead and run it and then check that it is running as expected.

```
# docker run -d -p 8500:8500 --name consul consul
```

So what have we just done;

- docker

This command just tells the terminal to invoke the docker application

- run

This command tells the docker application the activity we are going to action

- -d

This flag tells docker the container is going to run in detached mode, ie: we will run it and connect to it or interact with it in another way, do not drop us into the terminal.

- -p 8500:8500

This flag tells docker to map the host port:container port, so in this case, we explicitly want port 8500 on the host to map to 8500 of this container.

- --name consul

Every container can have a unique name. It is an easy way to address the container rather than remembering or looking up the Container ID each time.

- consul

This sets the container we are going to start. This is the name that you can see if you run "docker images". This will in most cases contain a /, but is not required to.

After running the above command you should receive a long number as your confirmation that the action was performed. It will look something like this;

```
> docker run -d -p 8500:8500 --name consul consul  
b0dc2607f1f29f6aeedf71bb4797b07f27c53bc94f898119b072a55329b336b
```

Lets check Docker correctly started the container and it is running as expected;

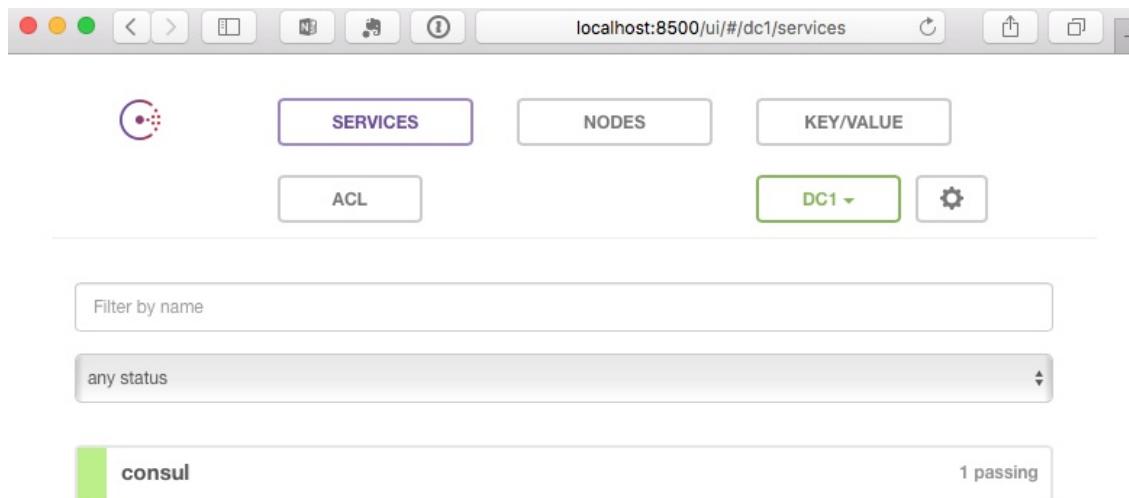
```
# docker ps -a
```

Step 2 - Consul Setup

You should see something like the below. Please note the ID number and mapped ports will be different.

CONTAINER ID	IMAGE	COMMAND	NAMES	CREATED	STATUS	PORTS
d07ffcee893a	consul	"docker-entrypoint.sh"	consul	3 seconds ago	Up 2 seconds	8300-8302/tcp, 8400/tcp, 8301-8302/udp, 8600/tcp, 8600/udp, 0.0.0.0:8500->8500/tcp

At this point a simple test should confirm you have started your first container and port mapping is working successfully! Open a web browser and navigate to <http://localhost:8500> and you should be presented with the following screen.



The final thing we need from Consul is it's IP address, if it is the first container, it is likley to be 172.17.0.2, but lets check;

```
# docker inspect -f '{{.Name}} - {{.NetworkSettings.IPAddress }}' $(docker ps -aq)
```

The output should be similar to the below;

```
> docker inspect -f '{{.Name}} - {{.NetworkSettings.IPAddress }}' $(docker ps -aq)
/consul - 172.17.0.2
```

Now we know the IP that has been assigned to our Consul container. Make a note of it as you will need it later in the lab.

Step 2 - Consul Setup

An alternative way to get this information is;

```
# docker inspect consul
```

The output will contain amongst other things the IP, it should be similar to the below;

```
"Gateway": "172.17.0.1",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"IPAddress": "172.17.0.2",
"IPPrefixLen": 16,
"IPv6Gateway": "",
"MacAddress": "02:42:ac:11:00:02",
```

Step 2 - Consul Setup - Troubleshooting

When you try to run the docker consul, you may receive a response like the following;

```
# docker run -d -p 8500:8500 --name consul consul
docker: Error parsing reference: "\u00ad" is not a valid repository/tag.
```

This is usually down to a port conflict. 8500 is a reserved port for Adobe ColdFusion, so if you have that installed, expect to receive the above error. Try stopping ColdFusion and running again. Alternatively Tunnelblick also seems to use port 8500, closing it also fixes this error.

If you do not have ColdFusion installed, try running the following commands and see what is using the port;

```
# netstat -anp tcp | grep 8500
tcp4      0      0  10.61.232.175.58500  a84-53-157-70.de.https ESTABLISHED
```

This means something is listening and so you can stop it. To find the PID of the process using the port, try the following;

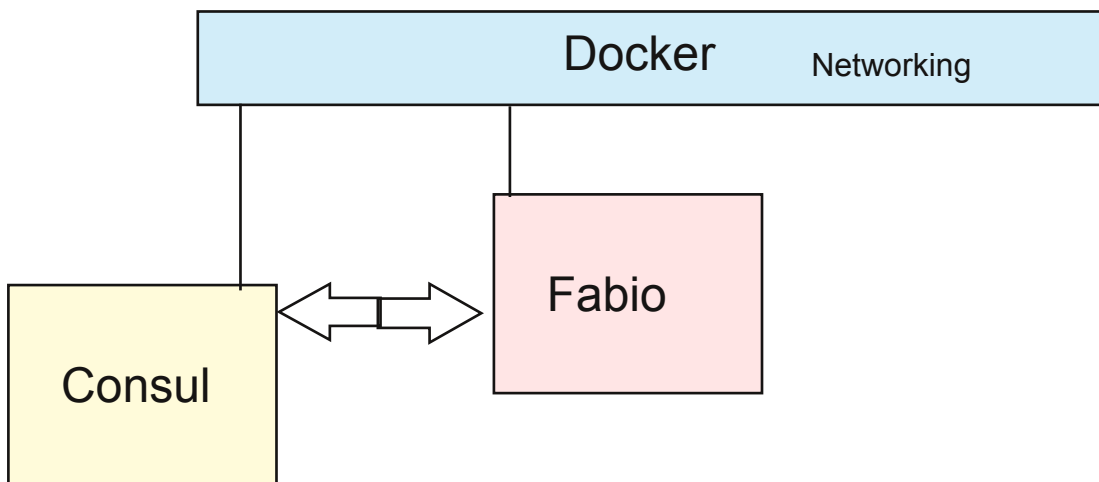
```
# lsof -i :8500
COMMAND PID  USER  FD  TYPE             DEVICE SIZE/OFF NODE NAME
firefox 5436 roporter  87u  IPv4 0xea8771add2a34547    0t0  TCP  ams-roporter-8919.cisco.com:59134->151.101.129.69:http (ESTABLISHED)
```

Then just kill the PID;

```
# kill 5436
```

Step 3

Fabio Setup



Step 3 - Fabio Setup

Fabio requires some configuration for our environment and we will achieve this via a simple text configuration file. There are many ways we could point fabio to the config file, but we will do this via sharing a folder in to the container. So the first thing we need is a shared folder.

```
# mkdir /tmp/lab/fabio
```

We have created it in the tmp folder to avoid any user access or privilege issues. Next, let's create our configuration file;

```
# nano /tmp/lab/fabio/fabio.properties
```

In this instance I am using nano to create the text file, however please use whichever text editor you are comfortable with and have installed on your system.

Paste the following into the file;

```
registry.consul.addr = <CONSUL IP>:8500
```

That's all we need in the config file and it is to ensure Fabio can correctly locate our Consul server. Finally we need to start our Fabio server, with the any ports we need and the config file we just created.

```
# docker run -d -p 9999:9999 -p 9998:9998 -v /tmp/lab/fabio:/etc/fabio --name fabio  
magiconair/fabio
```

So what have we just achieved;

-p 9999:9999

Map host port 9999 to container port 9999, this is the port we will access any service on.

-p 9998:9998

Map host port 9998 to container port 9998, this is the port we will access to view the services.

-v /tmp/lab/fabio:/etc/fabio

This is the folder map to ensure our config file is loaded by our Fabio server, essentially map local folder /tmp/lab/fabio to the container folder /etc/fabio.

--name fabio

Let's name the container to something that we can remember.

magiconair/fabio

This is the name of the container image we will run.

Step 3 - Fabio Setup

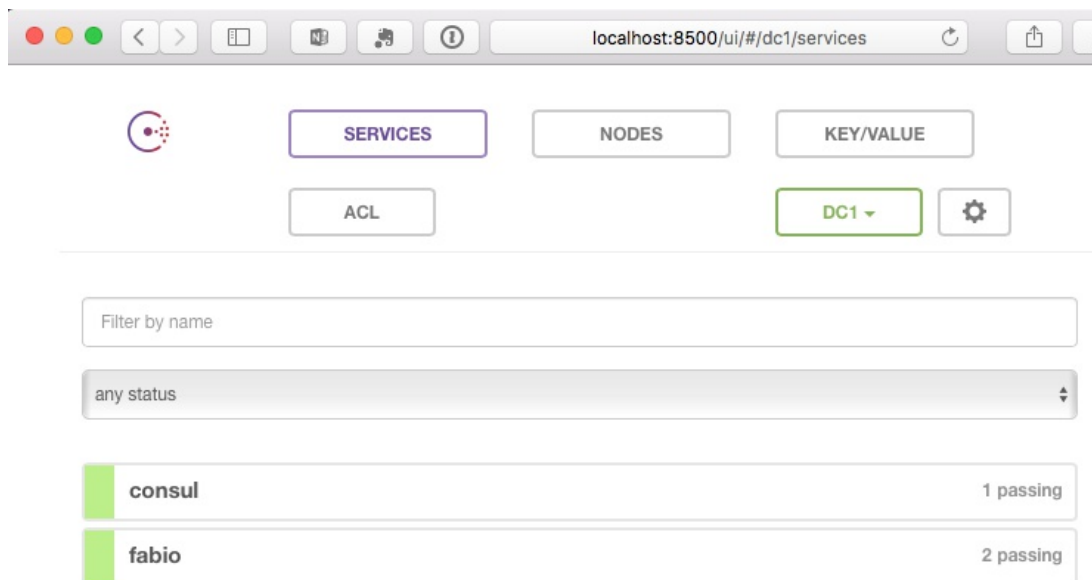
If everything worked successfully you should receive the long string back and running;

```
# docker ps -a
```

We should now see our two containers running. Checking the status column to ensure they are both up and running and neither have exited.

CONTAINER ID	IMAGE	COMMAND	NAMES	CREATED	STATUS	PORTS
6a459c48cf15	magiconair/fabio	"/fabio -cfg /etc/fab"	fabio	2 minutes ago	Up 2 minutes	0.0.0.0:9998-9999->9998-9999/tcp
09cbcf2693ae	consul	"docker-entrypoint.sh"	consul	4 minutes ago	Up 4 minutes	8300-8302/tcp, 8400/tcp, 8301-8302/udp, 8600/tcp, 8600/udp, 0.0.0.0:8500->8500/tcp

At this point our containers are up and running, now we need to check that Fabio has registered with Consul successfully. Open a web browser and go to <http://localhost:8500> and check the services tab.



Lastly we can browse to Fabio to ensure its ready for our application. Browse to <http://localhost:9998>



Step 3 - Fabio Setup - Troubleshooting

If you do not see Fabio listed on the on the services tab, there could be a number of things that are not working, however the most common is that Fabio was not able to read the configuration succesfully and that would usually be down to an incorrect folder mapping.

A quick way to check is to look at the config Fabio has, to do that, run the following command;

```
# docker logs fabio
```

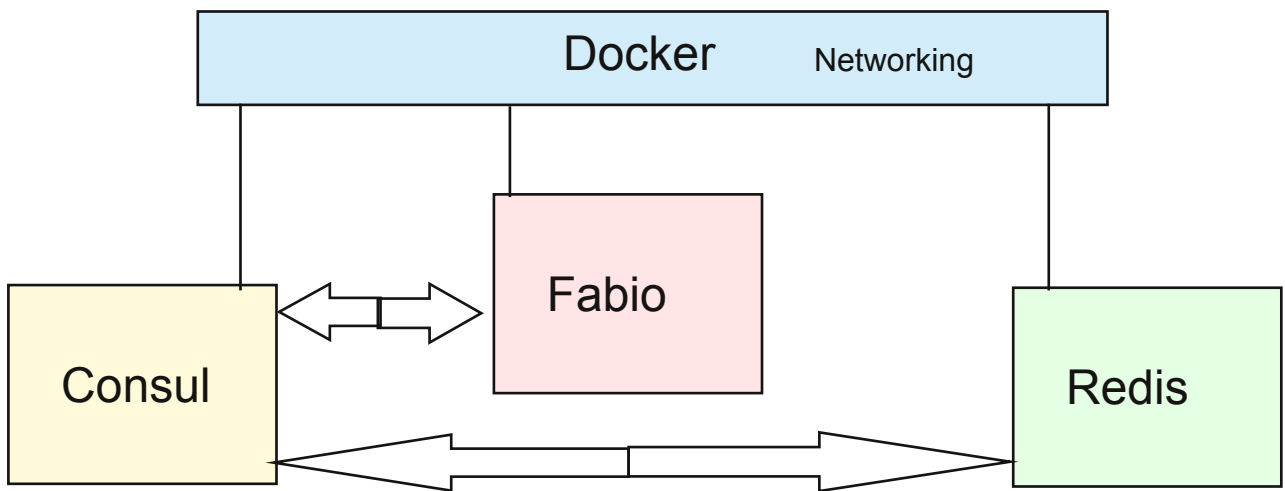
You may find some indication at the end of the log that tells you why or if the Fabio container is operating or not. If there are no obvious reasons, the Fabio config file will also be displayed. If you find the section in the config of Registry, then Consul and look for the Addr, this should map to the address you entered into the config file /tmp/lab/fabio/fabio.properties.

```
"Registry": {
  "Backend": "consul",
  "Static": {
    "Routes": ""
  },
  "File": {
    "Path": ""
  },
  "Consul": {
    "Addr": "172.17.0.2:8500",
    "Scheme": "http",
    "Token": "",
    "KVPath": "/fabio/config",
    "TagPrefix": "urlprefix-",
    "Register": true,
    "ServiceAddr": ":9998",
    "ServiceName": "fabio",
    "ServiceTags": null,
    "ServiceStatus": [
      "passing"
    ],
    "CheckInterval": 1000000000,
    "CheckTimeout": 3000000000
  },
}
```

If any of these details are incorrect, remove the container using "docker rm -f <ID>" and then try again, paying particular attention to the -v path of the docker run command.

Step 4

Redis Setup



Step 4 - Redis Setup

As we have already pulled the Redis image (This is not actually required. The following command will pull the image itself if it is not found on the local filesystem).

Running the following command will do everything we need for Redis.

```
# docker run -d -p 6379:6379 --name redis redis
```

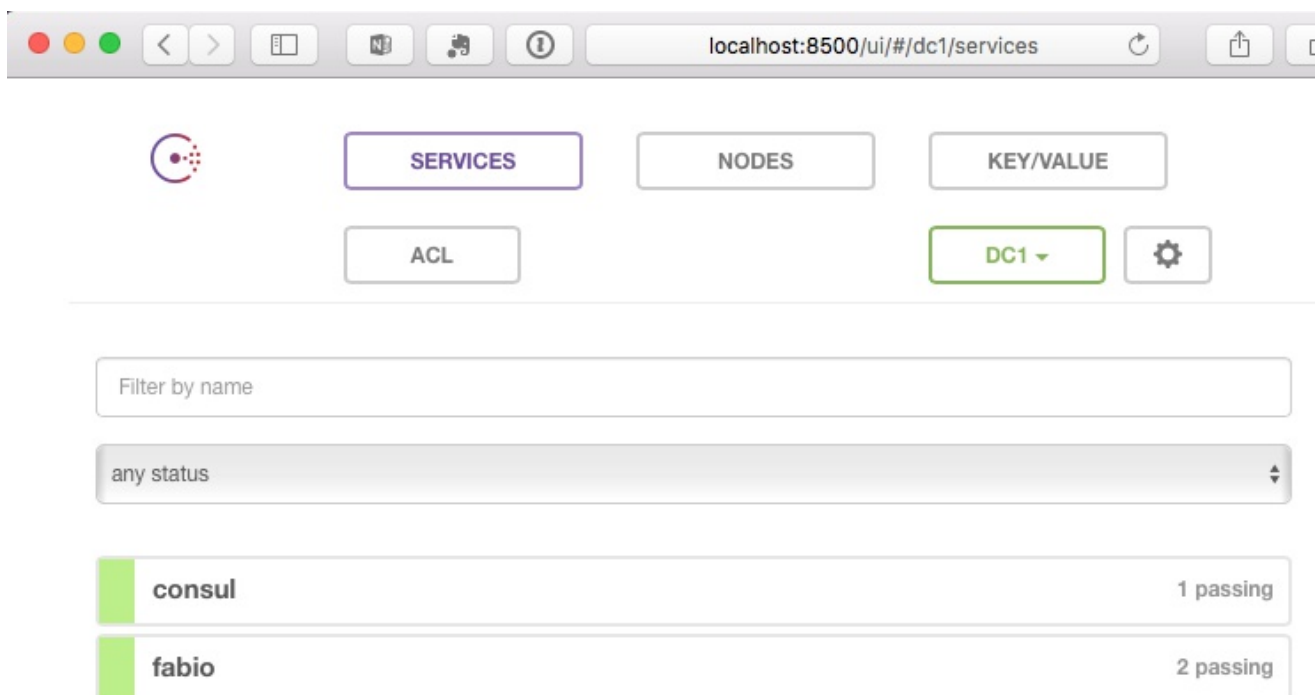
Lets test quickly to ensure it is running;

```
# docker ps -a
```

Hopefully the output now looks like the following;

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
930b6128cfeb	redis	"docker-entrypoint.sh"	22 hours ago	Up 18 hours	0.0.0.0:6379->6379/tcp	redis
6a459c48cf15	magiconair/fabio	"/fabio -cfg /etc/fab"	7 days ago	Up 5 days	0.0.0.0:9998-9999->9998-9999/tcp	fabio
09cbcf2693ae	consul	"docker-entrypoint.sh"	7 days ago	Up 5 days	8300-8302/tcp, 8400/tcp, 8301-8302/udp, 8600/tcp, 8600/udp, 0.0.0.0:8500->8500/tcp	consul

However if we check Consul it will not show our Redis server.

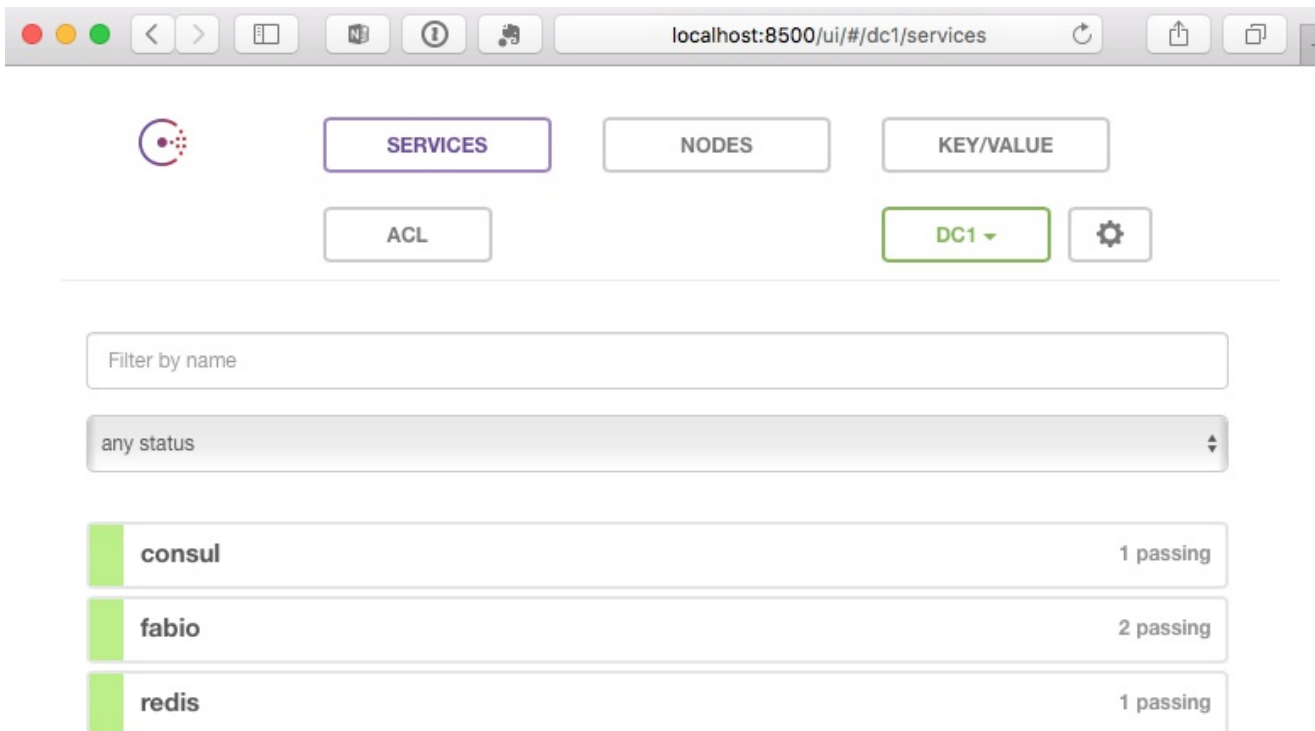


Step 4 - Redis Setup

As we will need to query Consul for the address of our Redis server we need to make Consul aware of it. There are different ways to achieve it, but using the default Redis image, we can just register it manually.

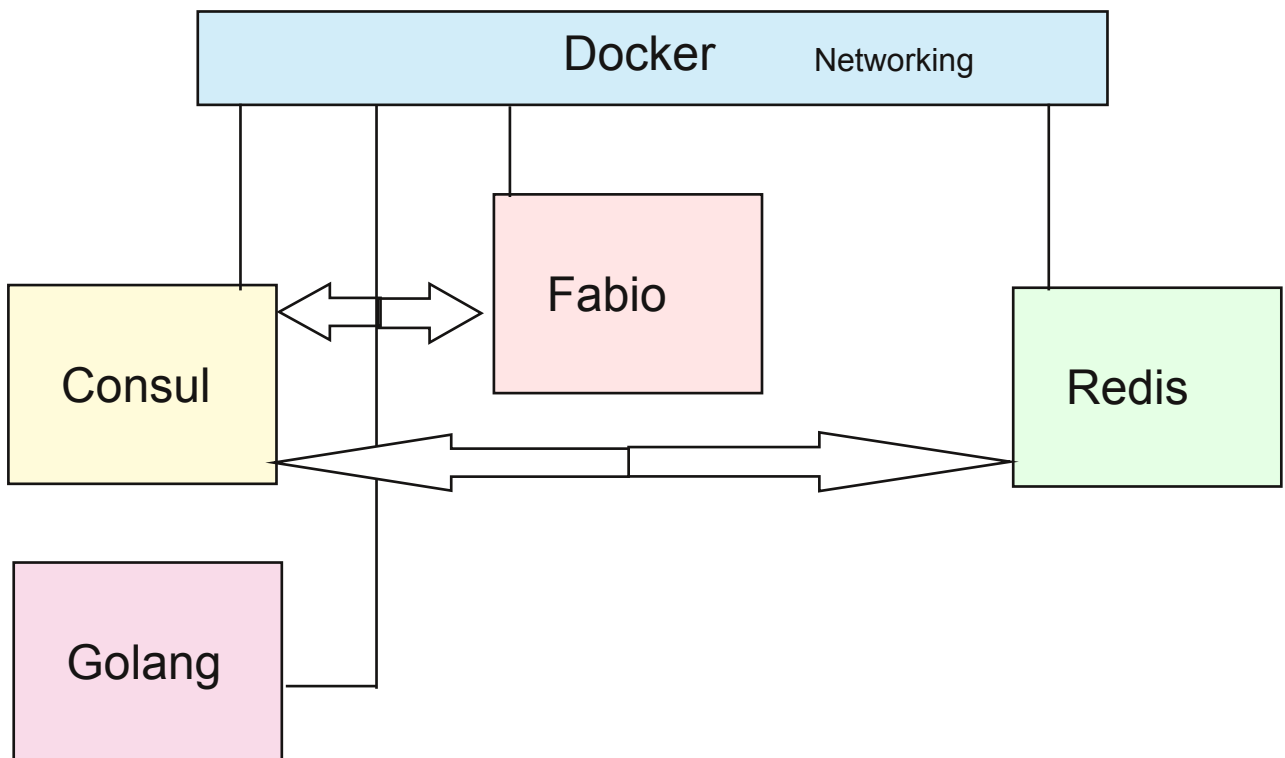
```
# curl -XPUT http://localhost:8500/v1/agent/service/register -d
'{"name":"redis","address":"<REDIS_IP>","port":6379}'
```

Nothing should happen at this point, but we can go to Consul and check to ensure that our Redis server now appears correctly.



Step 5

Golang Setup



Step 5 - Golang Setup

This section take us through installing and using Go within a container. If you have a local instance of GO then you could use it, but this is a simple and easy way to use a container for creating and building Go applications.

One of the great things about containers and one of the objectives is to try and keep the size to an absolute minimum. This helps with pushing and pulling and starting up new containers, Go can really help us here!

Go can compile the application and include all of the dependencies needed so the application can run on any system, this we will take advantage of with our simple webserver.

First off, open a new terminal window and then create a new folder for our work;

```
# mkdir /tmp/lab/golang
```

Lets create a very simple test case and ensure everything is working as expected.

```
# nano /tmp/lab/golang/main.go
```

Paste or copy in the following;

```
package main

import (
    "fmt"
    "net/http"
    "log"
)

func sayhello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "<html><head></head><body bgcolor='red'></body></html>")
}

func main() {
    http.HandleFunc("/", sayhello) // set router
    err := http.ListenAndServe(":8080", nil) // set listen port
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

Now we have our temporary folders and files created, lets start the Golang container, remember to do this in a new terminal window or tab;

```
# docker run -it --name golang -v /tmp/lab/golang:/tmp golang
```

This time we have started the container with different parameters and no ports mapped. The parameters I will explain next, but as we will be using this container purely as our GO compiler, we do not need any tcp/udp ports for remote connections.

Step 5 - Golang Setup

-it

This is the main difference when we run the golang container. This could also have been written as -i -t. These two parameters mean;

-i

Technically means keep the STDIN open always. However it is more commonly referred to as Interactive mode.

-t

Tells Docker to allocate a TTY to the container. This could be called a PTY, providing an emulation of a physical terminal.

--name

This is the same as all the other examples, we will specify a name, rather than having to

When you run this command, it will not return a container ID, it instead drops you at the terminal window for your container.

```
> docker run -it --name golang -v /tmp/golang:/tmp golang
root@33733b7dd51d:/go# |
```

To ensure all has worked, cd to /tmp and do an ls. You should see your main.go file you created earlier in this step.

```
root@33733b7dd51d:/go# cd /tmp
root@33733b7dd51d:/tmp# ls
main.go
root@33733b7dd51d:/tmp#
```

Lastly, just to ensure we can build our future code without errors, try to build our main.go file.

```
root@33733b7dd51d:/tmp# go build main.go
root@33733b7dd51d:/tmp# ls
main main.go
root@33733b7dd51d:/tmp#
```

Now we have our compiled binary for a very simple webserver we need some way of running it inside a container. This next example is a very simple way of doing it, but it has its issues which we will discuss later. Return to your main terminal window.

```
# nano /tmp/lab/golang/Dockerfile
```

First of all, we need to create a new file which will contain the information needed to build our very first customer container. The name of the file "Dockerfile" is fixed and needs to be entered exactly as is.

Step 5 - Golang Setup

Paste the following into the file;

```
FROM ubuntu
ADD main /
CMD ["/main"]
```

This is telling Docker to perform a few different actions;

FROM ubuntu

This means we will be using Ubuntu as our base image. So if the latest Ubuntu does not exist on the system, pull it from Docker hub.

ADD main /

As we have our compiled binary called "main", we want to copy it into the container so it can be executed.

CMD ["/main"]

Lastly, we are telling the container to execute this command, which in our case is the binary we copied in, in the previous command.

Now we need to build our new container;

```
# docker build -t robjporter/golang1 .
```

You can replace the "robjporter" piece with whatever you would like, this is for identification purposes only.

```
> docker build -t robjporter/golang1 .
Sending build context to Docker daemon 5.693 MB
Step 1 : FROM ubuntu
---> bd3d4369aebc
Step 2 : ADD main /
---> b0c745d9b363
Removing intermediate container 5ee0659cce22
Step 3 : CMD /main
---> Running in 211ad5bb6291
---> b1dd9d71ea0d
Removing intermediate container 211ad5bb6291
Successfully built b1dd9d71ea0d
```

If you saw the last message and it contained "Successfully built" you will now see your new container image listed with the others we have pulled for this lab. Trying listing the images;

```
> docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
robjporter/golang1	latest	b1dd9d71ea0d	3 seconds ago	132.3 MB

The size is pretty good, but our application is only 5.5MB, so why is it taking 132.3MB? There must be a better way and a more MicroService way of doing it?

Step 5 - Golang Setup

The last step in our testing is to run our new container. We do this in the same way as any other container.

```
> docker run -d -p 8080:8080 --name golang1 robjporter/golang1
df99a0707d726e7800b930a1f803a76c295b6c240b958e1f7e659bbbb0410b44
```

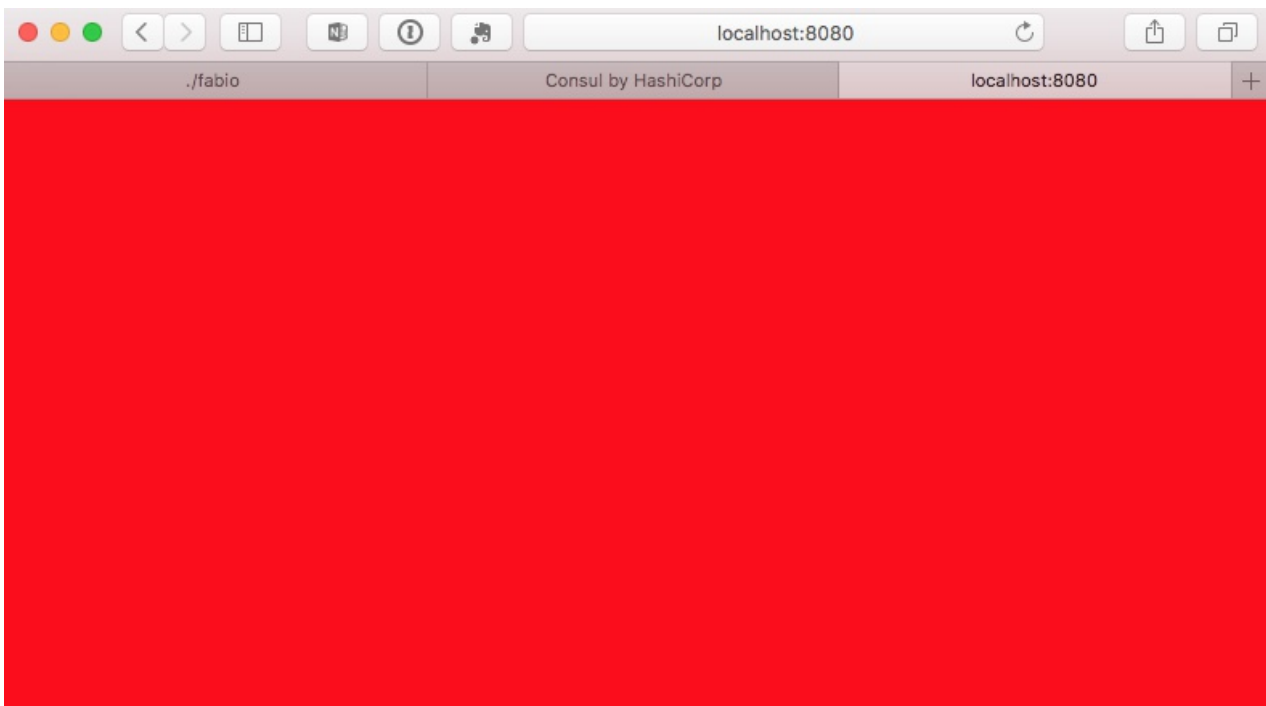
We have seen all of these parameters before, we are mapping localhost port 8080 to container port 8080, which is what we specided inside our GO application and we are referring to the container name, we used earlier, in my case "robjporter/golang1".

Lastly, lets check it is working as expected;

```
> docker ps -a
```

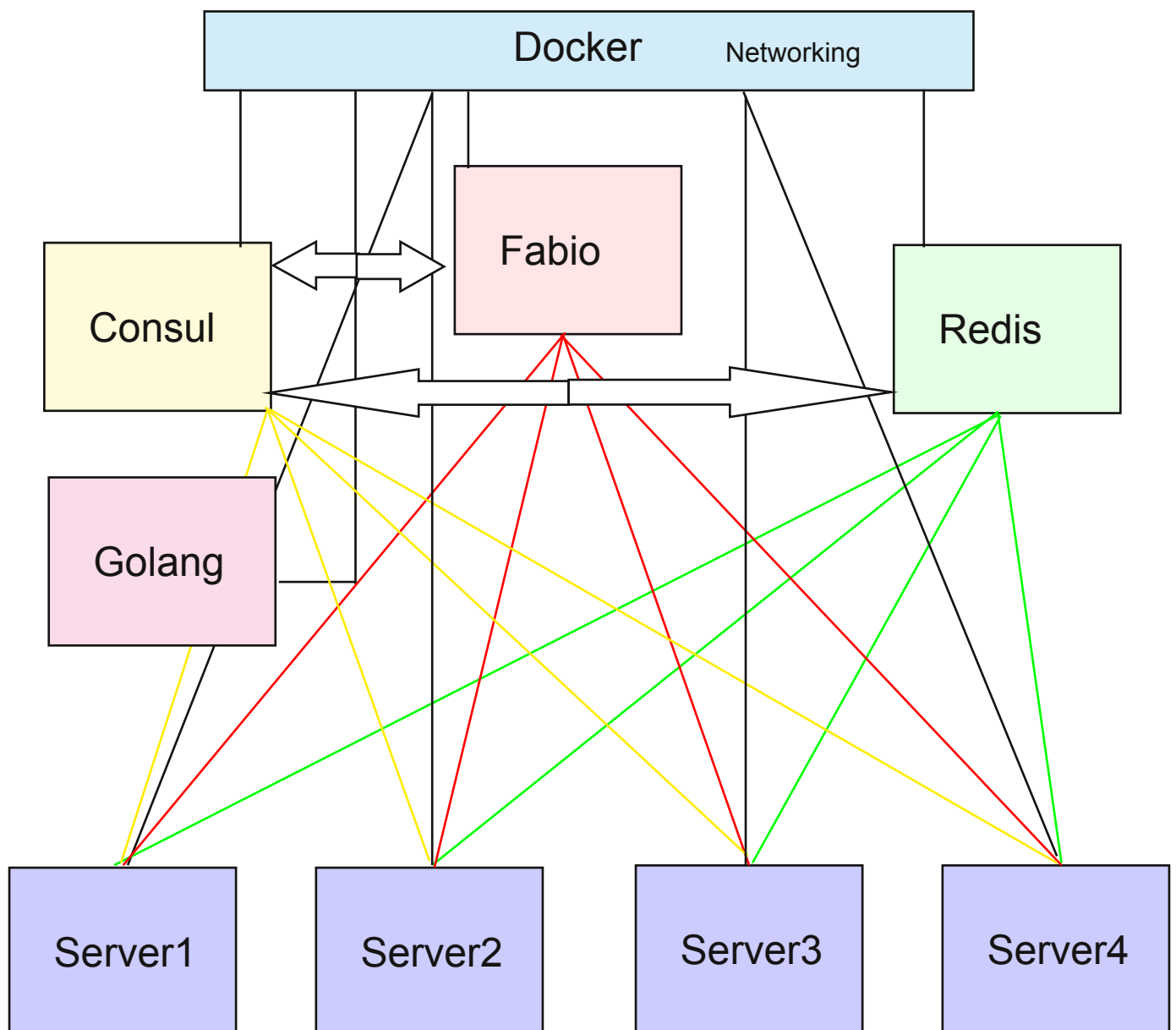
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
df99a0707d72	robjporter/golang1	"/main"	20 minutes ago	Up 20 minutes	0.0.0.0:8080->8080/tcp	golang1
33733b7dd51d	golang	"/bin/bash"	40 minutes ago	Up 40 minutes		golang
930b6128cfab	redis	"docker-entrypoint.sh"	36 hours ago	Up 36 hours	0.0.0.0:6379->6379/tcp	redis
6a459c48cf15	magiconair/fabio	"/fabio -cfg /etc/fab"	8 days ago	Up 6 days	0.0.0.0:9998-9999->9998-9999/tcp	fabio
09cbcf2693ae	consul	"docker-entrypoint.sh"	8 days ago	Up 6 days	8300-8302/tcp, 8400/tcp, 8301-8302/udp, 8600/tcp, 8600/udp, 0.0.0.0:8500->8500/tcp	consul

And;



Step 6

Bringing it all together



Step 6 - Bringing it all together

Now we have everything in place, tested and ready to build our main application.

What we have so far are a number of separate containers, services and applications, now is the time to join these together and make our Cloud Native Application take shape.

This lab will only touch on the beginnings of what a Cloud Native Application could look like and achieve, but it will leave it open for you to extend as you see fit. Plus if you are brave enough, some suggestions on what you could look at, add or adapt.

First of all, we need to remove the container we just created. As this was for test purposes, we will not longer need it.

```
# docker rm -f golang1
# docker rmi robjporter/golang1
```

Next we need to change the go file we created before, it is probably easier to create some new files and folders, so lets create four for now;

```
# mkdir /tmp/lab/golang/server1
# mkdir /tmp/lab/golang/server2
# mkdir /tmp/lab/golang/server3
# mkdir /tmp/lab/golang/server4
```

In each folder create a new Dockerfile;

```
# nano /tmp/lab/golang/server1/Dockerfile
# nano /tmp/lab/golang/server2/Dockerfile
# nano /tmp/lab/golang/server3/Dockerfile
# nano /tmp/lab/golang/server4/Dockerfile
```

And paste the content on the following page into each Dockerfile;

```
FROM scratch
ADD main /
CMD ["/main"]
EXPOSE 8080
```

This is very similar to the previous example, but just to explain;
FROM scratch

Scratch is a special Docker image which is empty, infact the size of image is actually 0 bytes. Which means anything you need for the image needs to be provided or be a native executable.

ADD main / and CMD ["/main"]

Are the same as before and just move our executable over and ensure it is started with the container.

EXPOSE 8080

This tells Docker that the container will use port 8080 and can dynamically map a host port to the container port 8080.

Step 6 - Bringing it all together

The next thing we need to do is to create a main.go file in each of the server folders too. The content you need to paste in is on the following page. The only thing you need to change or adapt is the variable "<COLOUR>", my suggestion would be to use colours that are very different, so red, blue, green and yellow for example.

```
# nano /tmp/lab/golang/server1/main.go
# nano /tmp/lab/golang/server2/main.go
# nano /tmp/lab/golang/server3/main.go
# nano /tmp/lab/golang/server4/main.go
```

As a quick note, due to our new servers now registering with Consul, we need to ensure we correctly deregister our servers.

If we used the normal command "docker -rm f server1", it will remove the container, however in Consul, the server will leave remnants in the services list and Consul will be left waiting for it to return, which may never happen.

Therefore we need to correctly send a termination signal to the server, so it correctly shuts down and removes itself from Consul before terminating the container.

To achieve this, whenever you wish to stop one of the webserver, please use the following command;

```
# docker kill --signal=SIGINT <NAME>
```

This kills or stops the container, but it does it by sending a signal to the container OS. Our application is listening for this signal and will take appropriate actions, ie: unregister itself from Consul.

Step 6 - Bringing it all together

```
package main

import (
    "fmt"
    "net/http"
    "github.com/robjporter/CPVT-CloudNativeLab/lab"
)

var serverCount string
var dbStartCount string

func sayhello(w http.ResponseWriter, r *http.Request) {
    color := "<COLOUR>"
    content := "<html><head></head><body bgcolor=\"" + color + "\">"
    content += serverCount + " servers currently serving web content.<br>"
    content += dbStartCount + " server starts have been seen.<br>"
    content += lab.GetPageCount() + " page loads have happened.<br>"
    content += "</body></html>"
    fmt.Fprintf(w, content)
}

func healthCheck(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "OK\n")
}

func main() {
    http.HandleFunc("/", sayhello) // set router
    http.HandleFunc("/health", healthCheck) // set router

    urlsToRegister := []string{"/"}
    ipOfConsul := "<IP>"
    portWeListenOn := "8080"

    result, err := lab.RegisterMe(ipOfConsul, urlsToRegister, portWeListenOn)
    serverCount = lab.GetServerCount("localhost-")
    dbStartCount = lab.GetDBStartCount()

    if result {
        fmt.Println("Server started and listening on port :"+portWeListenOn)
        err = http.ListenAndServe(":"+portWeListenOn, nil) // set listen port
        if err != nil {
            fmt.Println("ListenAndServe: ", err)
        }
    } else {
        fmt.Println(err)
    }
}
```

Step 6 - Bringing it all together

For those not familiar with GO or wanting some more detail on what the code you are pasting in is doing, this is break down;

```
import (
    "fmt"
    "net/http"
    "github.com/robjporter/CPVT-CloudNativeLab/lab"
)
```

These are just the GO modules we need for our application to function. The first two are standard libraries, with the third being one I wrote specifically for the Cloud VT.

```
var serverCount string
var dbStartCount string
```

These are some public variables that we will be using in multiple functions.

```
func sayhello(w http.ResponseWriter, r *http.Request) {
    color := "<COLOUR>"
    content := "<html><head></head><body bgcolor=\"" + color + "\">"
    content += serverCount + " servers currently serving web content.<br>"
    content += dbStartCount + " server starts have been seen.<br>"
    content += lab.GetPageCount() + " page loads have happened.<br>"
    content += "</body></html>"
    fmt.Fprintf(w, content)
}
```

This is our main web page that users will be accessing. The function is passed in a reference and pointer, they refer to the input http request (r) and the output http response (w).

In the function we define a color variable, it has been separated out, as we will change it for each server, all it does is change the background color, so we can differentiate between servers.

We build up the content variable, which is what we finally write out on the last line to the http response output.

The content is made up of a few lines, mostly to identify and prove we have a working service.

The serverCount variable queries Consul for the number of active webserver we have running.

The dbStartCount variable queries our redis server for how many times a webserver has started.

The lab.GetPageCount() is a call to get from redis how many page loads have happened in total.

Step 6 - Bringing it all together

```
func healthCheck(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w,"OK\n")
}
```

This is a very important function as it is called by Consul to ensure that our webserver is active and ready for receiving connections. Although it is very simple, it responds with a 200 http status code and OK in the body.

```
func main() {
    http.HandleFunc("/", sayhello) // set router
    http.HandleFunc("/health", healthCheck) // set router

    urlsToRegister := []string{"/"}
    ipOfConsul := "<IP>"
    portWeListenOn := "8080"

    result, err := lab.RegisterMe(ipOfConsul, urlsToRegister, portWeListenOn )
    serverCount = lab.GetServerCount("localhost-")
    dbStartCount = lab.GetDBStartCount()

    if result {
        fmt.Println("Server started and listening on port :"+portWeListenOn)
        err = http.ListenAndServe(":"+portWeListenOn, nil) // set listen port
        if err != nil {
            fmt.Println("ListenAndServe: ", err)
        }
    } else {
        fmt.Println(err)
    }
}
```

This is the main function for our application and is essentially used to setup all the bits we need and start the webserver to start listening for connections.

The first two lines of the function are setting up our URL paths, this means our webserver will listen for connections coming into / and /health, when they are matched in the URL it will call the corresponding function.

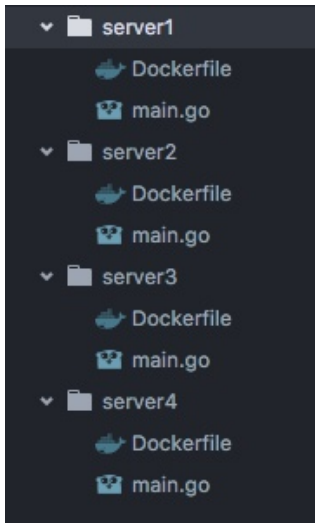
The next three lines are some variables we need for registration into Consul. The first one is a simple a list of the URL's that our servers care about, in this case it is only /. The other two are hopefully self explanatory.

The next four lines are getting some data from the custom package built for the lab, the first registers our server in Consul, the second gets the number of servers registered in Consul and the third gets some data from our redis server.

The last section has only one purpose and that is to start the webserver listening if everything else was successful.

Step 6 - Bringing it all together

You should now have a structure very similar to the following.



```
root@33733b7dd51d:/tmp# ls
Dockerfile lab main main.go server1 server2 server3 server4
root@33733b7dd51d:/tmp# ls -la
total 5580
drwxr-xr-x 11 root root    374 Oct  5 15:05 .
drwxr-xr-x 45 root root   4096 Oct  5 07:38 ..
-rw-r--r--  1 root root   6148 Oct  5 10:22 .DS_Store
-rw-r--r--  1 root root    37  Oct  5 07:56 Dockerfile
drwxr-xr-x  5 root root    170 Oct  5 10:07 lab
-rwxr-xr-x  1 root root 5689643 Oct  5 07:54 main
-rw-r--r--  1 root root    402 Oct  5 07:54 main.go
drwxr-xr-x  4 root root    136 Oct  5 14:59 server1
drwxr-xr-x  4 root root    136 Oct  5 15:00 server2
drwxr-xr-x  4 root root    136 Oct  5 15:00 server3
drwxr-xr-x  4 root root    136 Oct  5 15:01 server4
root@33733b7dd51d:/tmp#
```

We now need to build our server application. Go back to the terminal connected to the golang container and run;

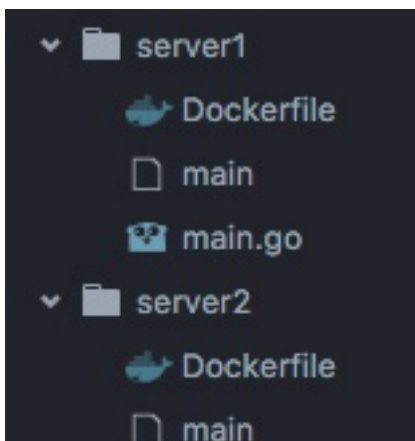
```
golang# go get -u gopkg.in/redis.v4
golang# go get -u github.com/robjporter/CPVT-CloudNativeLab/lab
```

Once the packages have been downloaded we are ready to build the executable, which we will do in a special way. Repeat this for each server;

```
golang# cd server1
golang# CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
```

```
root@33733b7dd51d:/tmp# cd server1
root@33733b7dd51d:/tmp/server1# CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
root@33733b7dd51d:/tmp/server1# cd ..
root@33733b7dd51d:/tmp# cd server2
root@33733b7dd51d:/tmp/server2# CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
root@33733b7dd51d:/tmp/server2# cd ..
root@33733b7dd51d:/tmp# cd server3
root@33733b7dd51d:/tmp/server3# CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
root@33733b7dd51d:/tmp/server3# cd ..
root@33733b7dd51d:/tmp# cd server4
root@33733b7dd51d:/tmp/server4# CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
root@33733b7dd51d:/tmp/server4#
```

Once you have built the four servers, you could now close the terminal window that you have been using for connecting to the golang container as it is no longer needed. You can also go ahead and remove the container.



Step 6 - Bringing it all together

We now need to build our servers so that we can then start them from a Docker image;

```
# docker build -t robjporter/server1 server1/
# docker build -t robjporter/server2 server2/
# docker build -t robjporter/server3 server3/
# docker build -t robjporter/server4 server4/
```

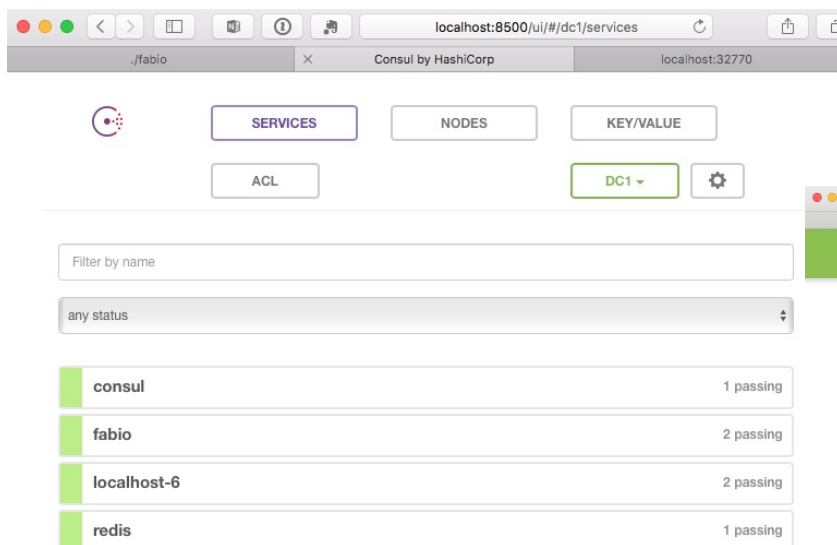
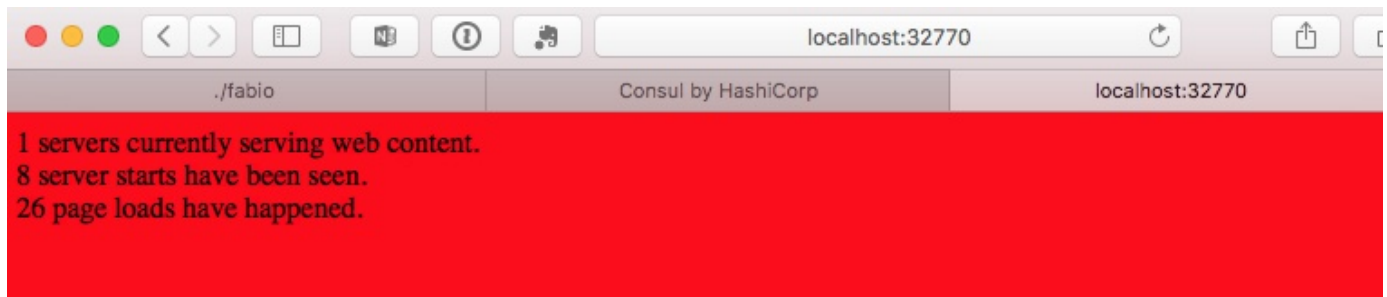
If all has gone well, when you now list your docker images, you will see there are now four new images and they are now very small roughly 6.5MB in size!

```
> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
robjporter/server4   latest             353d30af109f       About a minute ago 6.424 MB
robjporter/server3   latest             d1a23e7e6769       2 minutes ago      6.424 MB
robjporter/server2   latest             79174b447a15       2 minutes ago      6.424 MB
robjporter/server1   latest             2276131376e1       2 minutes ago      6.424 MB
```

Finally we need to start our servers, lets try one first

```
# docker run -d -P --name server1 robjporter/server1
```

To ensure this has worked successfully, please check Consul and Fabio to ensure they can see your server, you can also open a webbrowser to the mapped port;



Routing Table

Type to filter routes

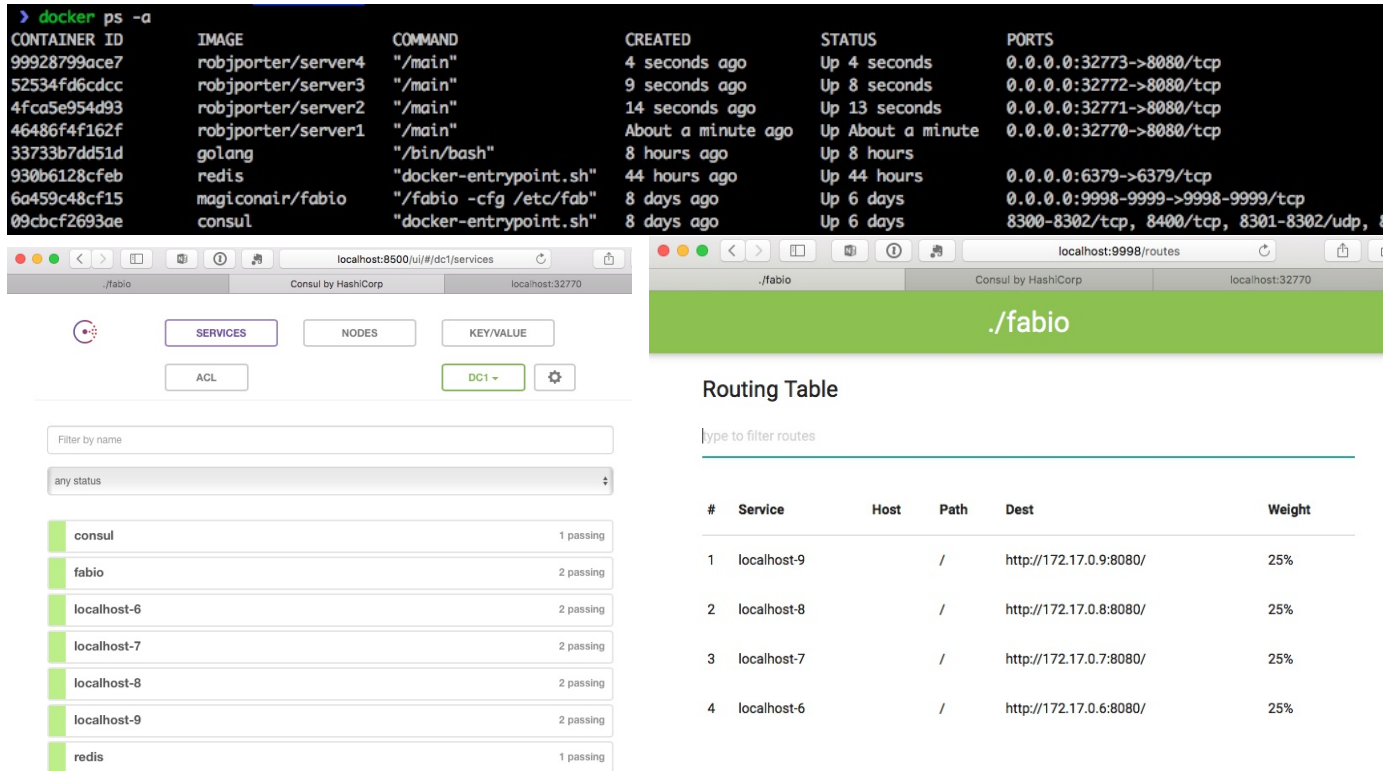
#	Service	Host	Path	Dest	Weight
1	localhost-6	/		http://172.17.0.6:8080/	100%

Step 6 - Bringing it all together

Now start the remaining servers;

```
# docker run -d -P --name server2 robjporter/server2
# docker run -d -P --name server3 robjporter/server3
# docker run -d -P --name server4 robjporter/server4
```

And check everything has worked as expected;

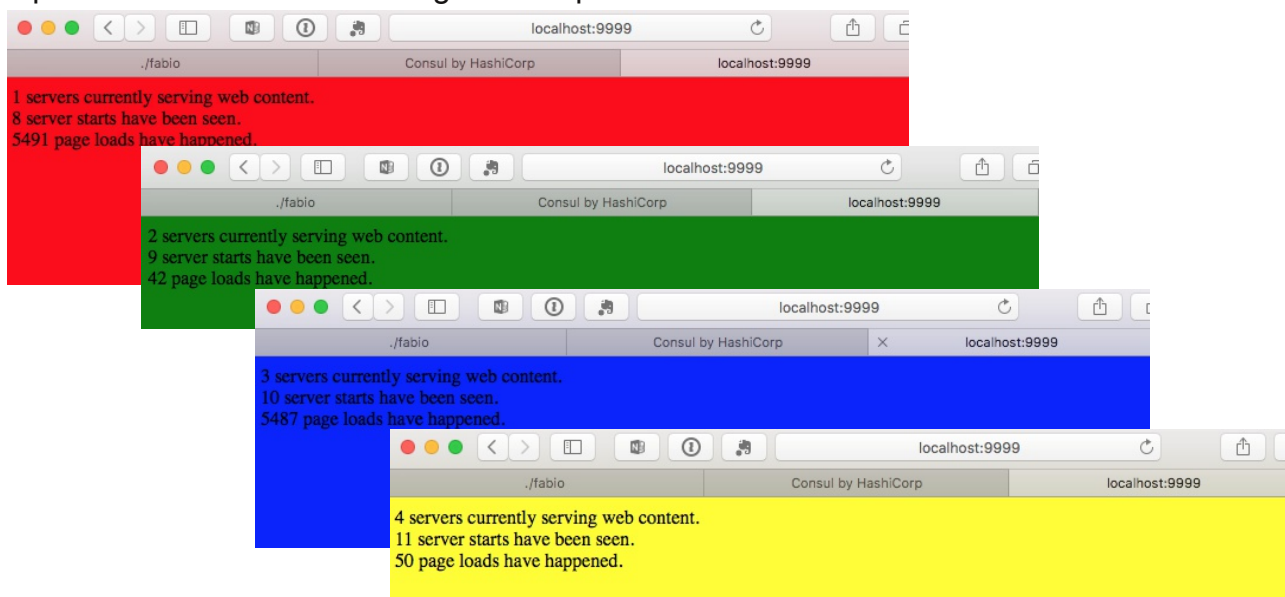


The screenshot shows the output of the command `docker ps -a` and the Consul UI. The Docker output shows four containers (server1, server2, server3, server4) running the `robjporter/server1` image, and two containers (fabio, consul) running the `magiconair/fabio` image. The Consul UI shows the services page with a list of services: consul, fabio, localhost-6, localhost-7, localhost-8, localhost-9, and redis. The Routing Table shows the following routes:

#	Service	Host	Path	Dest	Weight
1	localhost-9		/	http://172.17.0.9:8080/	25%
2	localhost-8		/	http://172.17.0.8:8080/	25%
3	localhost-7		/	http://172.17.0.7:8080/	25%
4	localhost-6		/	http://172.17.0.6:8080/	25%

If you have the above output, then the lasy thing is to test it out properly!

Open a web browser and navigate to <http://localhost:9999>

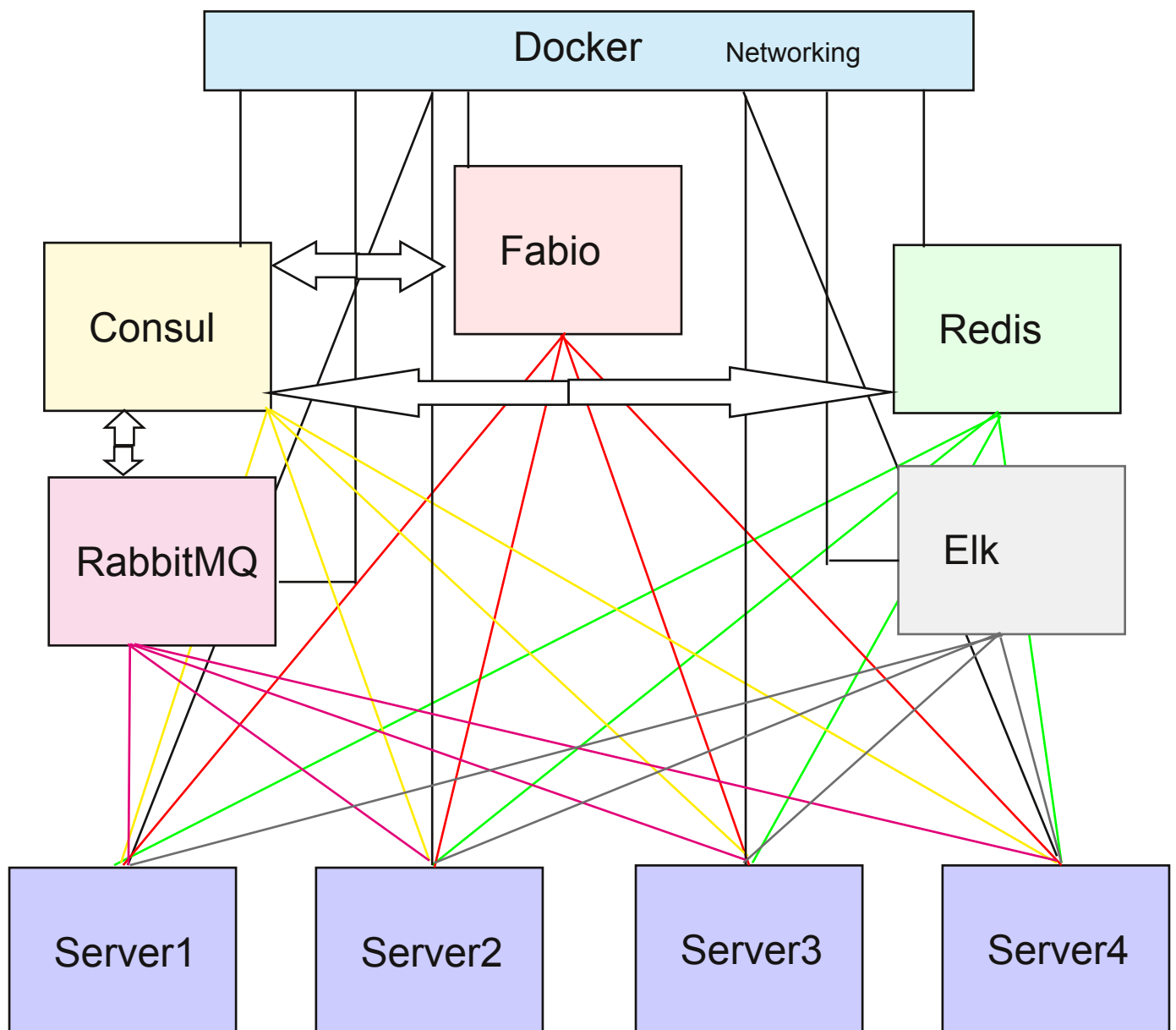


The screenshot shows a web browser displaying the Fabio status page. The page shows the number of servers currently serving web content, the number of server starts, and the number of page loads. The status is updated as more servers are added:

- 1 servers currently serving web content. 8 server starts have been seen. 5491 page loads have happened.
- 2 servers currently serving web content. 9 server starts have been seen. 42 page loads have happened.
- 3 servers currently serving web content. 10 server starts have been seen. 5487 page loads have happened.
- 4 servers currently serving web content. 11 server starts have been seen. 50 page loads have happened.

Step 7

Where to now



Step 7 - Where to now!

Congratulations!!!!

If you made it this far, you will have a great and fully working, containerised, load balanced, self discovering application and a good start to a full microservice.

So what else could you add?

Well depending on how brave you are, try some of these;

- 1) Have a look at running Consul in a cluster, rather than as a single node. Have a look at some of the parameters for the Consul server, maybe `-bind`, `-client` and `-retry-join` will be good starting points?
- 2) Try changing the HTML, I built the webserver to increment and show the actual order the servers started in and how many servers had started at that time, maybe you want to display different information?
- 3) Try installing Hey and see what load you can point towards `localhost:9999`, <https://github.com/rakyll/hey>. A small tip is keep the consecutive small, 25 is good, but you may need to adjust as everything is hosted on your one laptop.
- 4) Try stopping your webserver and checking the behaviour, what do you expect to happen? what actually happens?
- 5) Maybe you could add logging and monitoring, try adding ELK and seeing if you can send some log information to it?
- 6) Try adding a message queue. Add a new URL to each webserver /queue and when it is called it adds a message onto the queue. Don't forget something has to take things off of the queue and action them. RabbitMQ could be a good choice.
- 7) Try attaching to containers, why do some provide an output and some do not allow you to attach?
- 8) Try executing some commands on the different containers, for example, get consul to list the current Consul members.
- 9) Try adding in Vault <https://www.vaultproject.io/>, keeping some secret data in it and getting each server to read from it when it starts and then display it on the home html screen.

Step 7 - Where to now!

Running Consul in a cluster

The following commands are for guidance only and you should check you IP addresses before using;

```
# docker run -dp 8500:8500/tcp --name=consul1 consul agent -dev -ui -client 172.17.0.2 -bind 172.17.0.2
# docker run -d --name consul2 consul agent -dev -retry-join=172.17.0.2 -bind 172.17.0.3
# docker run -d --name consul3 consul agent -dev -retry-join=172.17.0.2 -bind 172.17.0.4
```

Adding a message queue

First we need to pull rabbitmq

```
# docker pull rabbitmq
```

Then we need to start it;

```
# docker run -d -p 5672:5672 --hostname rabbitmq --name rabbitmq rabbitmq
```

We will need port 5672 so we will be able to send jobs to rabbitMQ. The only new thing we have included is the hostname parameter. This is a simple measure for how rabbitMQ stores information internally and you could use a different name if you want to.

We also need to register it with Consul, so we will do something similar to what we did with Redis;

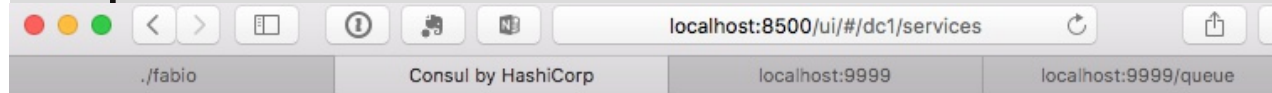
```
# curl -XPUT http://localhost:8500/v1/agent/service/register -d '{"name":"rabbitmq","address":"<IP>","port":5672}'
```


Please note, the name parameter needs to be set exactly as is in lower case and the "<IP>" needs replacing with the IP of your rabbitMQ server. Please remember, this can be gained from inspecting the container.

Now using the same processes as before, replace the servers 1-4 by building new servers from the code in servers1.1-4.1.

The following screenshots will give you an impression of what your environment should look like, once you have completed the above steps.

Step 7 - Where to now!





SERVICES

NODES

KEY/VALUE

ACL

DC1 ▾

⚙️

Filter by name

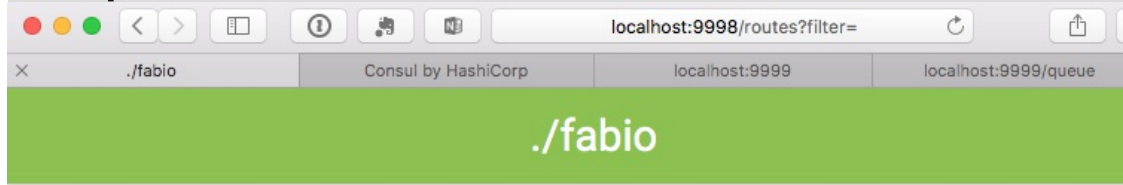
any status ▾

consul	1 passing
fabio	2 passing
localhost-11	2 passing
localhost-5	2 passing
localhost-6	2 passing
localhost-7	2 passing
rabbitmq	1 passing
redis	1 passing



Message sent to Queue for processing

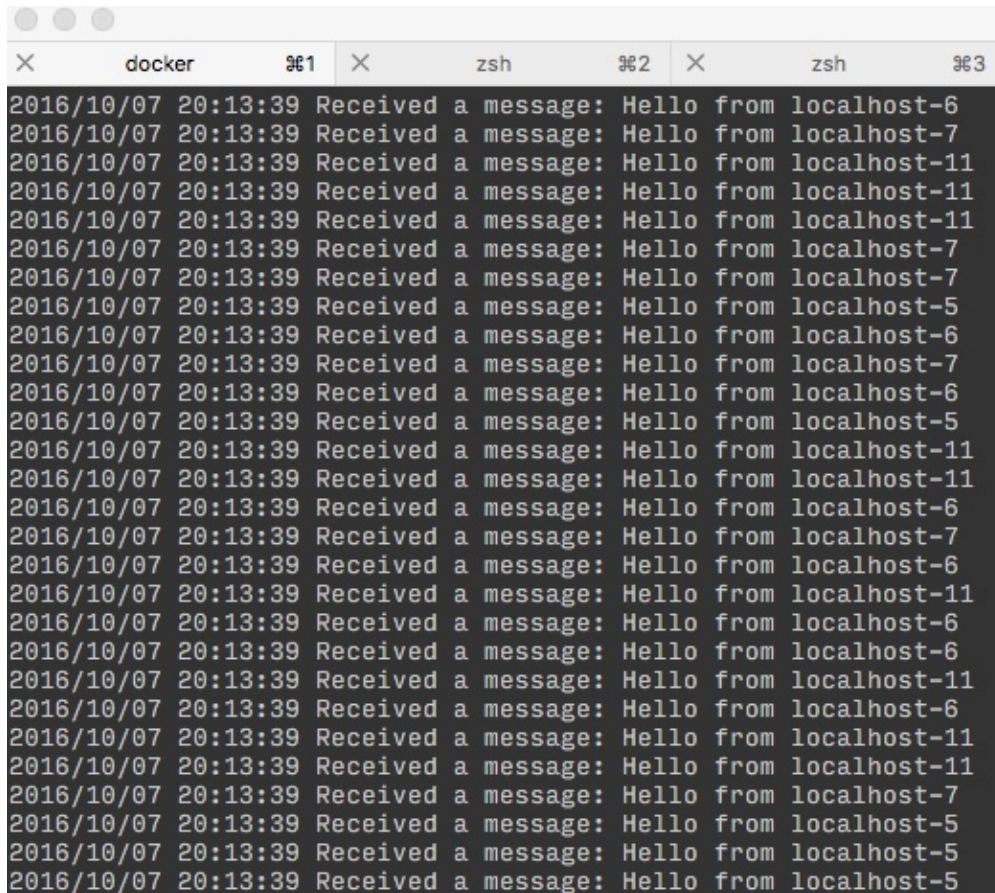
Step 7 - Where to now!



Routing Table

type to filter routes

#	Service	Host	Path	Dest	Weight
1	localhost-7		/queue	http://172.17.0.7:8080/	25%
2	localhost-6		/queue	http://172.17.0.6:8080/	25%
3	localhost-5		/queue	http://172.17.0.5:8080/	25%
4	localhost-11		/queue	http://172.17.0.11:8080/	25%
5	localhost-7		/	http://172.17.0.7:8080/	25%
6	localhost-6		/	http://172.17.0.6:8080/	25%
7	localhost-5		/	http://172.17.0.5:8080/	25%
8	localhost-11		/	http://172.17.0.11:8080/	25%



Step 8 - Other things to try

The following is just a list of other tools you should try;

UI For Docker

```
# docker run -d -p 9000:9000 --privileged -v /var/run/docker.sock:/var/run/docker.sock uifd/ui-for-docker
```

Navigate to <http://localhost:9000>

cAdvisor

```
# docker run -d -p 8080:8080 -v /:/rootfs:ro -v /var/run:/var/run:rw -v /sys:/sys:ro -v /var/lib/docker:/var/lib/docker:ro --name=cadvisor google/cadvisor:latest
```

Navigate to <http://localhost:8080>



CISCO