Cloud Native Application Lab

About

The lab is originally created by Rob Porter of Cisco Systems. This Lab is a copy of the original lab with some improvements.

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.

Lab requirements

There are only two requirements for this lab;

1. A running and installed version of Docker. The Lab is tested with Docker for Windows (on Windows 10 Pro), Docker Toolbox on Windows 7 and on Mac OS X. You should 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. An active internet connection to enable you to download some images from Docker Hub

Lab Components

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 flavors 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

http://www.github.com

GitHub is a webbased 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.

Docker

http://www.docker.com

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.

Consul

http://www.consul.io

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

Fabio

https://github.com/eBay/fabio

fabio is a fast, modern, zeroconf 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.

Redis

http://www.redis.io

Redis is an open source (BSD licensed), inmemory 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.

Golang

http://golang.org

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.

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 assumption 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.

The commands for MAC and other *nix platforms are equal. For Windows users the commands can differ. In the document you will find the commands for both platforms.

Note 1: if you are running on Windows 7 and therefore have Docker Toolbox installed, you will need to find the IP of the Ubuntu server, 192.168.99.100 is most likely please replace localhost with this IP throughout the manual.

Note 2: Please use the Chrome browser if possible throughout the lab.

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

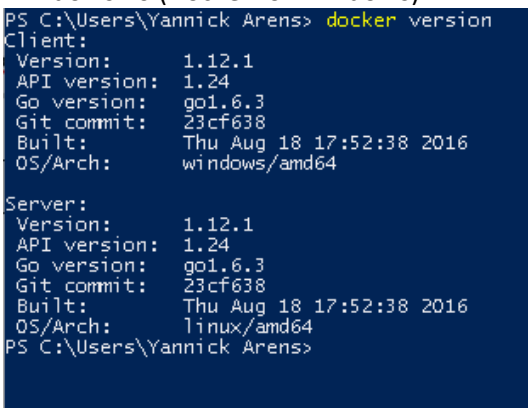
Step 1a: Check Docker version

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 from your terminal screen (Unix) or CMD/PowerShell (Windows);

```
# docker version
```

Unix

Windows 10 (Docker for Windows)



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 specifics.

Step 2b: Clean up environment

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
```

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

Unix:

Windows:

Now check the available images;

```
# docker images
```

Unix:

Windows:

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 1c: Download images

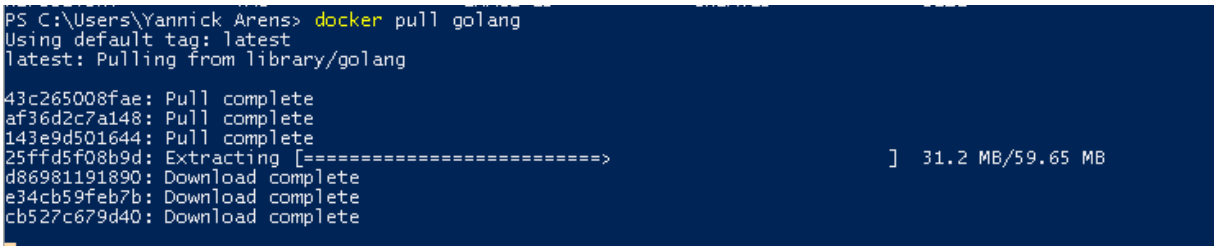
We can now pull a couple of images we will need for this lab. Please be patient, it will take a few minutes for each of these commands to complete;

```
# docker pull golang
```

During this process you should see output that resembles;

Unix:

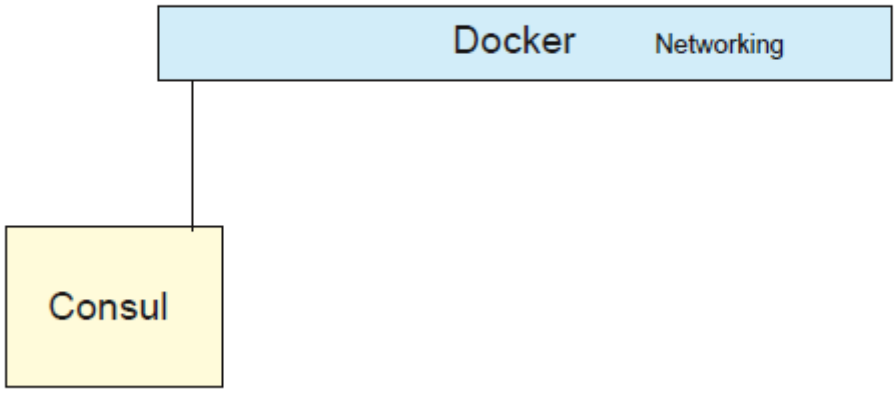
Windows:



Continue and pull the other four images we need;

```
# docker pull consul

# docker pull redis

# docker pull magiconair/fabio

# docker pull ubuntu
```

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;

Unix:

```
 ❯ docker run -d -p 8500:8500 --name consul consul
b0dc2607f1f29f6aeeedf71bb4797b07f27c53bc94f898119b072a55329b336b
```

Windows:

```
PS C:\Users\Yannick Arens> docker run -d -p 8500:8500 --name consul consul
715b5aa2f4d79334b7e41be0421a7e1574eed98f83f5e412513bcc13b42cfc5e
```

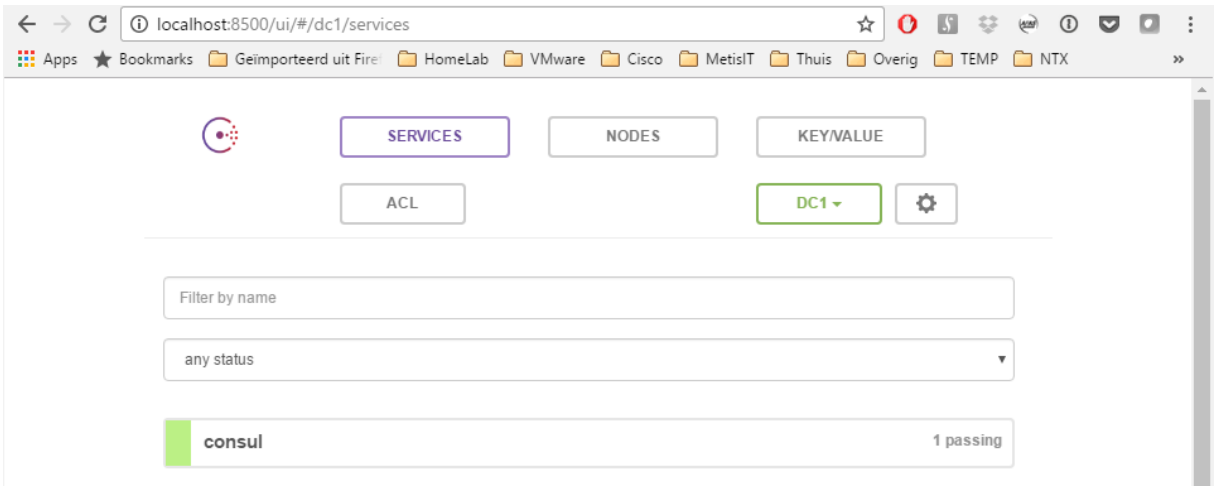
Let's check Docker correctly started the container and it is running as expected;

```
# docker ps -a
```

Unix/Windows:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
                                                                                     NAMES
715b5aa2f4d7        consul              "docker-entrypoint.sh"   2 hours ago         Up 3 minutes        8300-8302/tcp,
8400/tcp, 8301-8302/udp, 8600/tcp, 8600/udp, 0.0.0.0:8500->8500/tcp    consul
```

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 (or the IP you noted if you are using Docker toolbox) 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 likely to be 172.17.0.2, but let's check;

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

The output should be similar to the below;

Unix/Windows:

```
PS C:\Users\Yannick Arens> docker inspect -f '{{.Name}} - {{.NetworkSettings.IPAddress }}' $(docker ps -aq)
/consul - 172.17.0.2
```

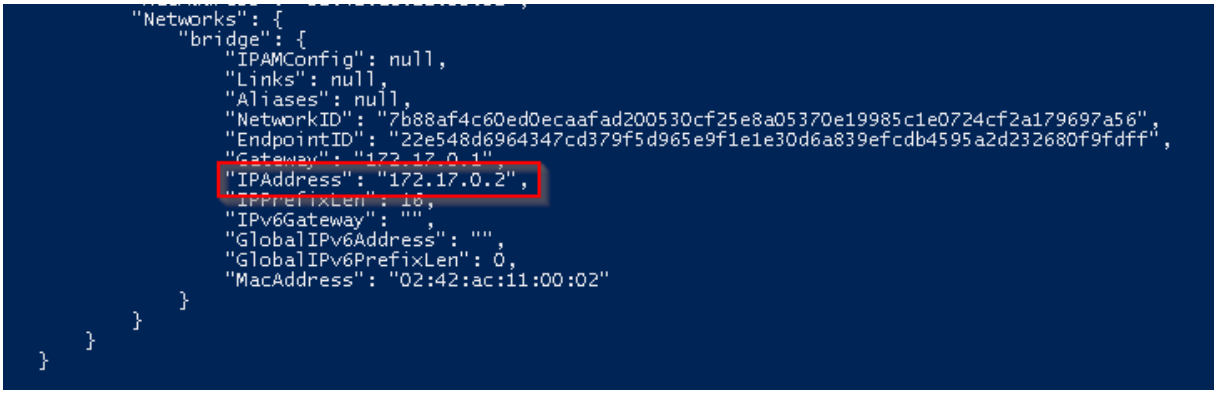
Now we know the IP that has been assigned to our Consul container. Make a note of it as you will need it later in the lab.

An alternative way to get this information is;

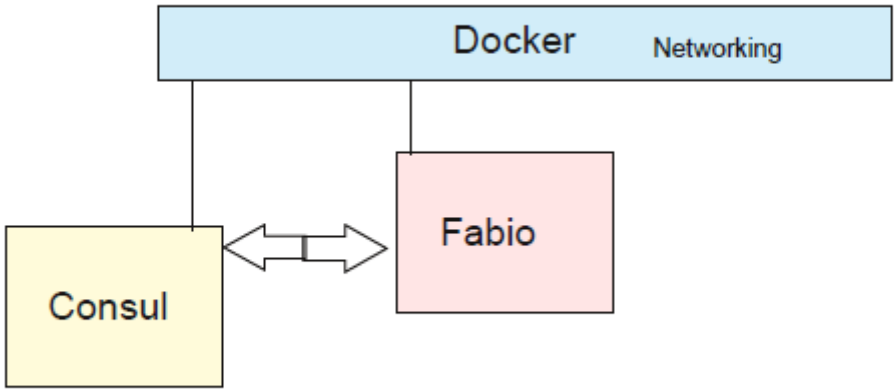
```
# docker inspect consul
```

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

Unix/Windows:



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.

Unix:

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

We have created it in the tmp folder to avoid any user access or privilege issues.

Windows*:

```
# mkdir 'c:/Users/<<Username>>/tmp/lab/fabio'
```

*Make the directory in your user directory because this directory will be accessible in your docker environment.

Next, lets create our configuration file;

Unix:

```
# 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.

Windows:

Use your own preferred text editor to create this file in the fabio folder, like notepad++.

Paste the following into the file;

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

Please replace <CONSUL IP> with the IP of you consul server, from the previous step. Please also note, if you are on a windows machine, it might have added ".txt" file extension. Please remove that before proceeding.

Thats 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.

Unix:

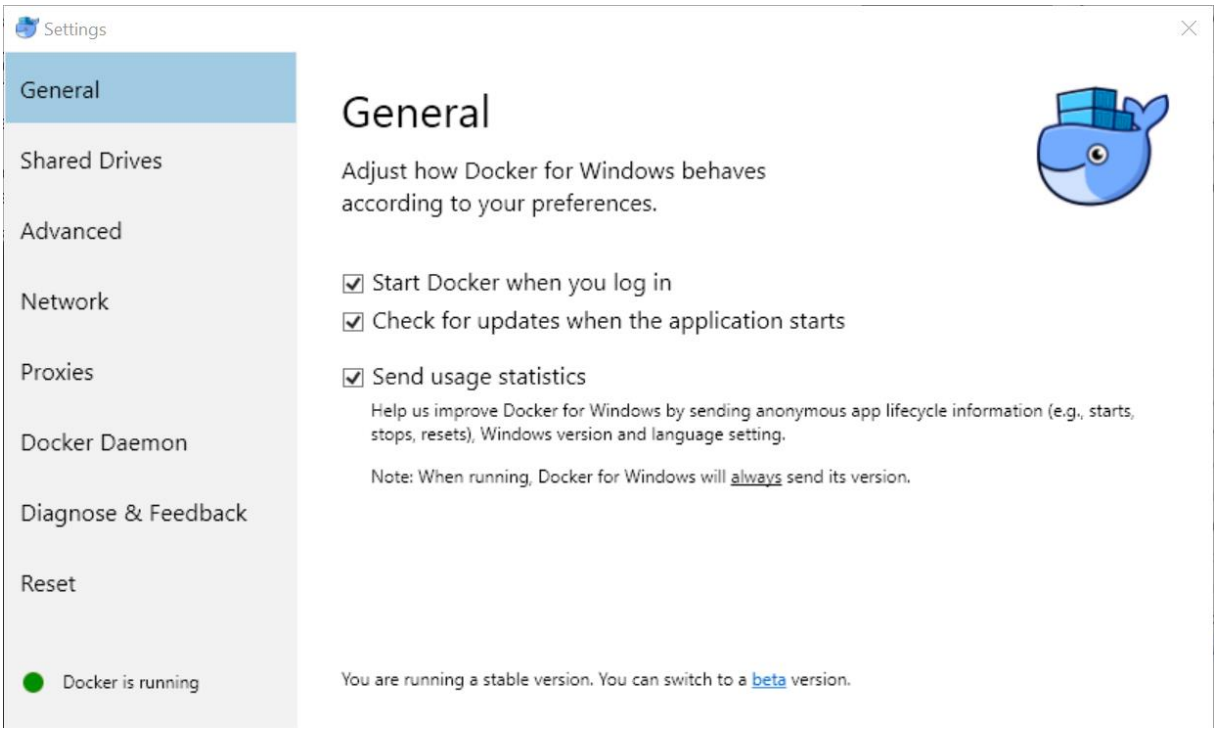
```
# docker run -d -p 9999:9999 -p 9998:9998 -v /tmp/lab/fabio:/etc/fabio --name fabio magiconair/fabio
```

Windows:

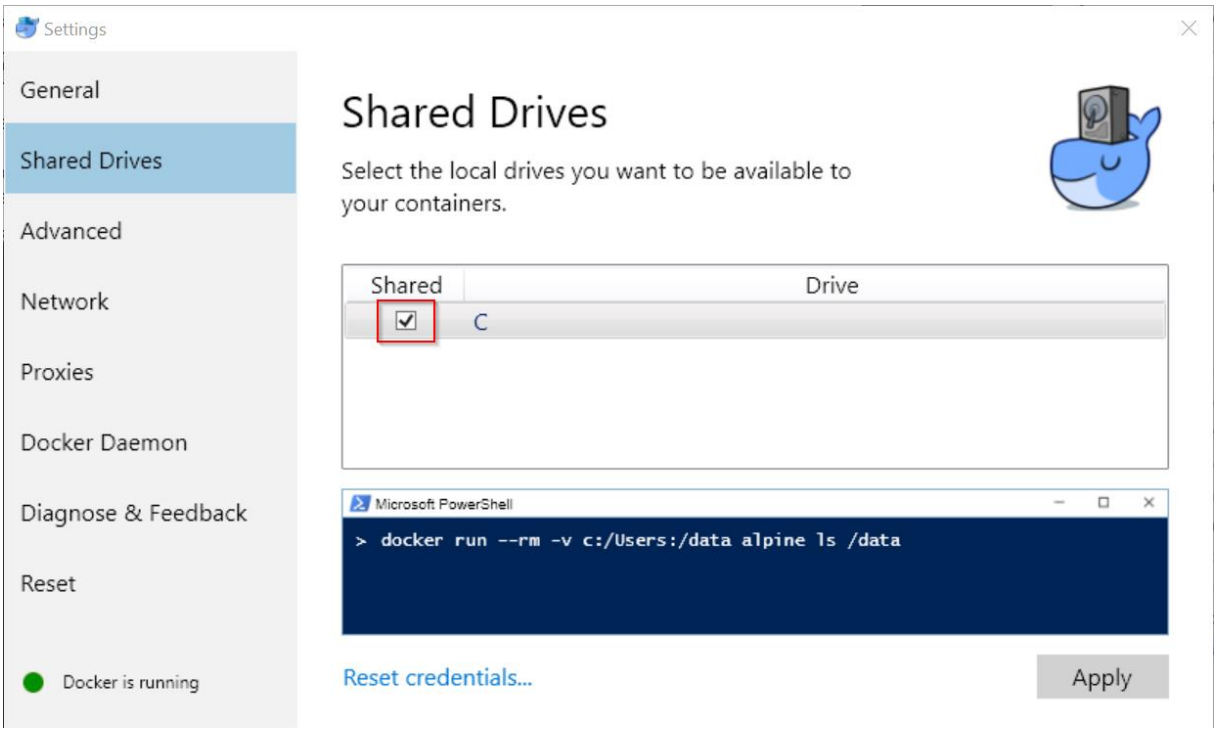
Before you can do this step you need to enable the shared drives option within Docker.

Go to your Docker daemon taskbar

Right click and click Settings.



Select Shared Drives and select the checkmark. Fill in your Windows credentials and click on Apply.



After this you can run the commands below.

```
# docker run -d -p 9999:9999 -p 9998:9998 -v
'/c/Users/<<USERNAME>>/tmp/lab/fabio:/etc/fabio' --name fabio magiconair/fabio
```

Replace <<USERNAME>> for your username.

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 '/c/Users/<<USERNAME>>/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	Lets name the container to something that we can remember.
Magiconair/Fabio	This is the name of the container image we will run.

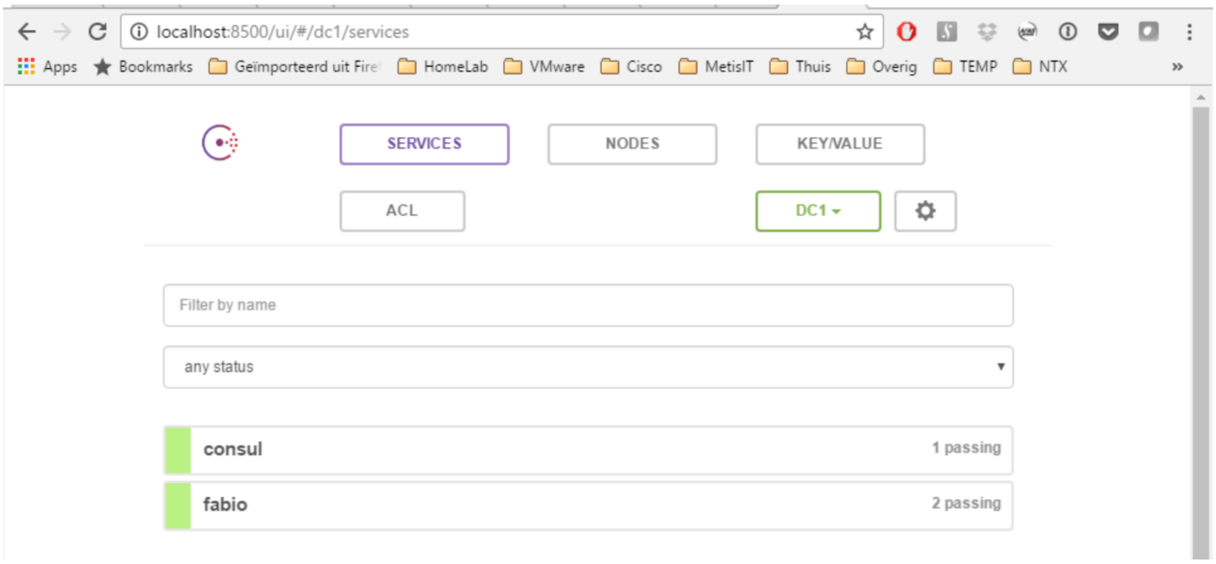
If everything worked successfully you should recieve the long string back and running;

```
# docker ps -a
```

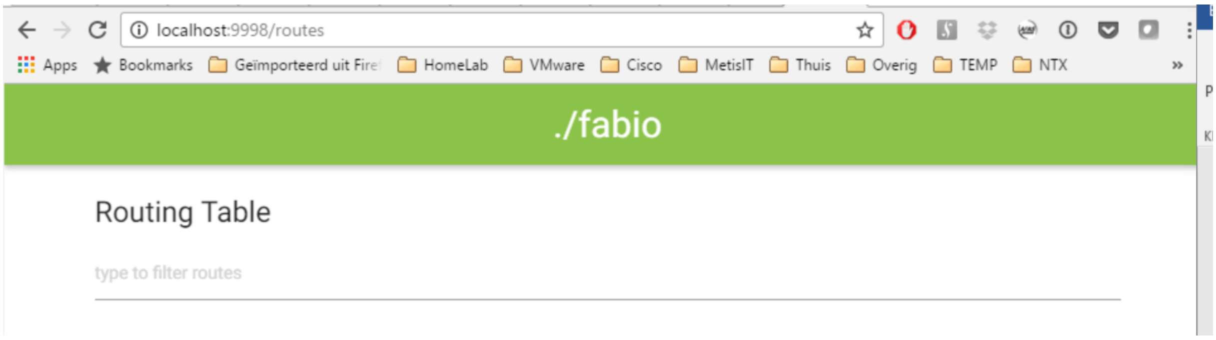
We should now see our two containers running. Checking the status column to ensure they are both up and running and neither have exited.

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
                                                                      NAMES
d58a033a6095        magiconair/fabio    "/fabio -cfg /etc/fab"   2 hours ago         Up 22 minutes       9999/tcp, 0.0.0
.0:9999->9998/tcp                                                      fabio
715b5aa2f4d7        consul              "docker-entrypoint.sh"   3 hours ago         Up About an hour    8300-8302/tcp,
8400/tcp, 8301-8302/udp, 8600/tcp, 8600/udp, 0.0.0.0:8500->8500/tcp   consul
```

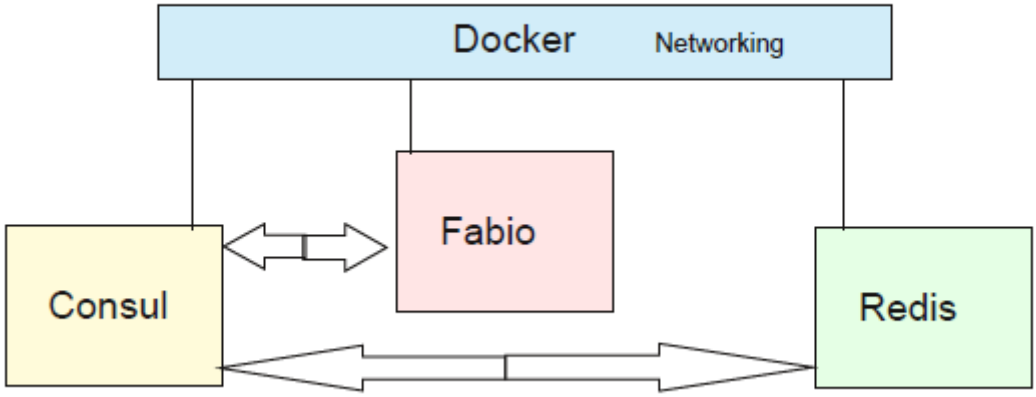
At this point our containers are up and running, now we need to check that Fabio has registered with Consul successfully. Open a web browser and go to http://localhost:8500 (or the IP you noted if you are using Docker toolbox) and check the services tab.



Lastly we can browse to Fabio to ensure its ready for our application. Browse to http://localhost:9998 (or the IP you noted if you are using Docker toolbox).



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
```

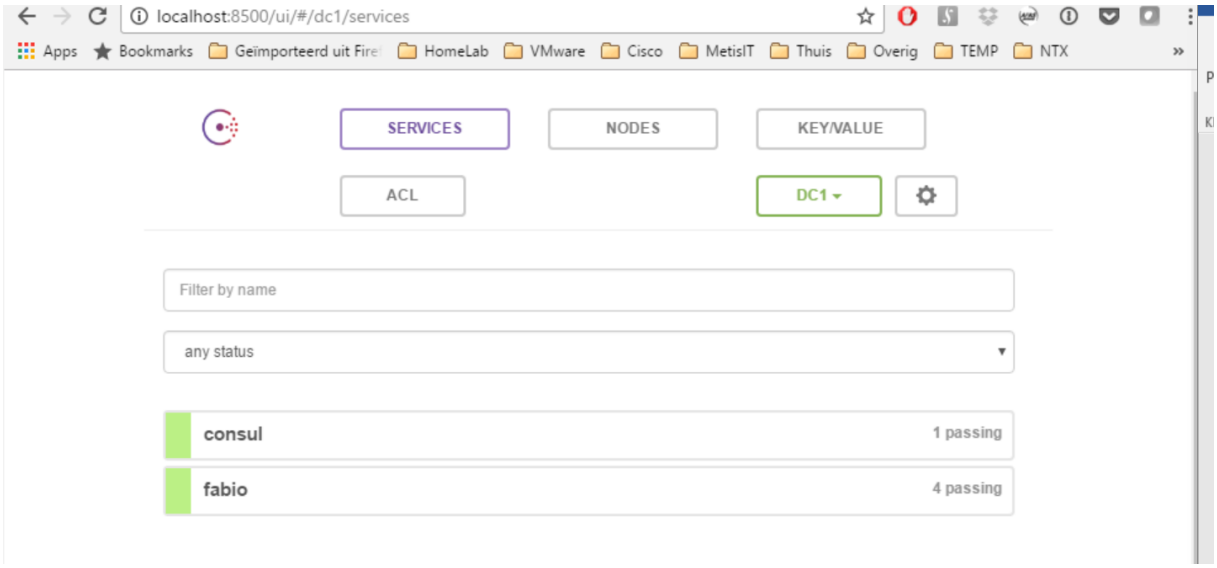
Let's test quickly to ensure it is running;

```
# docker ps -a
```

Hopefully the output now looks like the following;

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                                                                                  NAMES
b211be5eaaa7        redis               "docker-entrypoint.sh"   2 hours ago         Up 3 minutes        0.0.0.0:6379->6379/tcp                                                                 redis
ffe6afd29ff3        magiconair/fabio    "/fabio -cfg /etc/fab"   2 hours ago         Up 7 minutes        0.0.0.0:9998-9999->9998-9999/tcp                                                       fabio
715b5aa2f4d7        consul              "docker-entrypoint.sh"   3 hours ago         Up About an hour    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.



Please run Docker Inspect to get the IP for your redis server.

```
# docker inspect redis
```

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.

Unix:

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

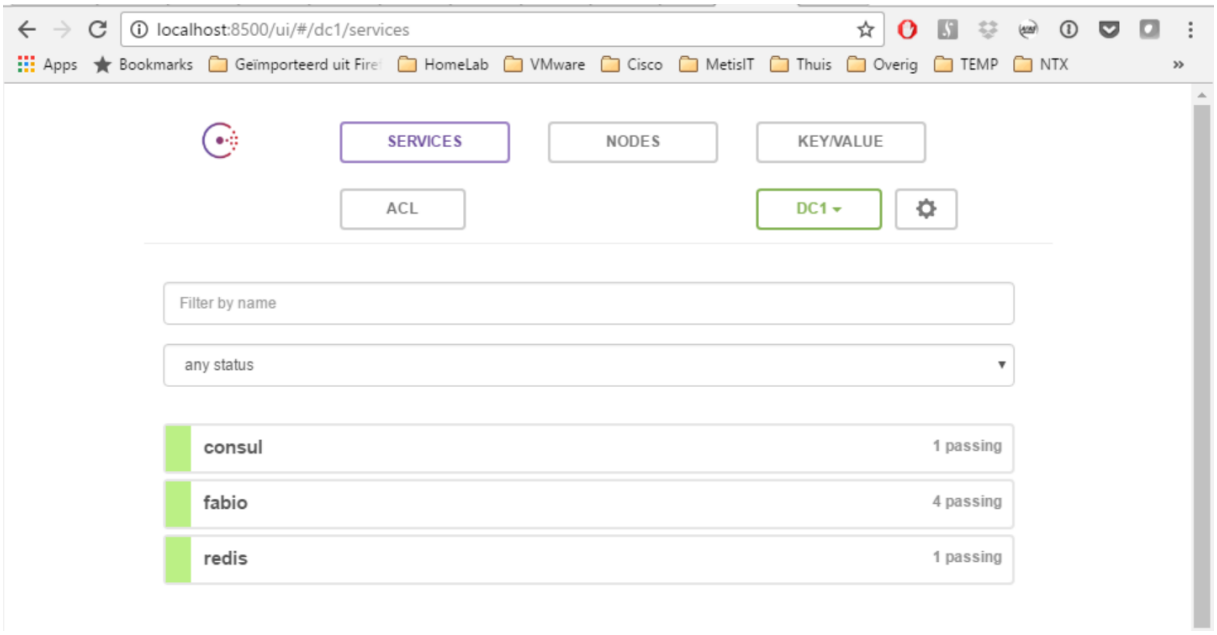
Windows:

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

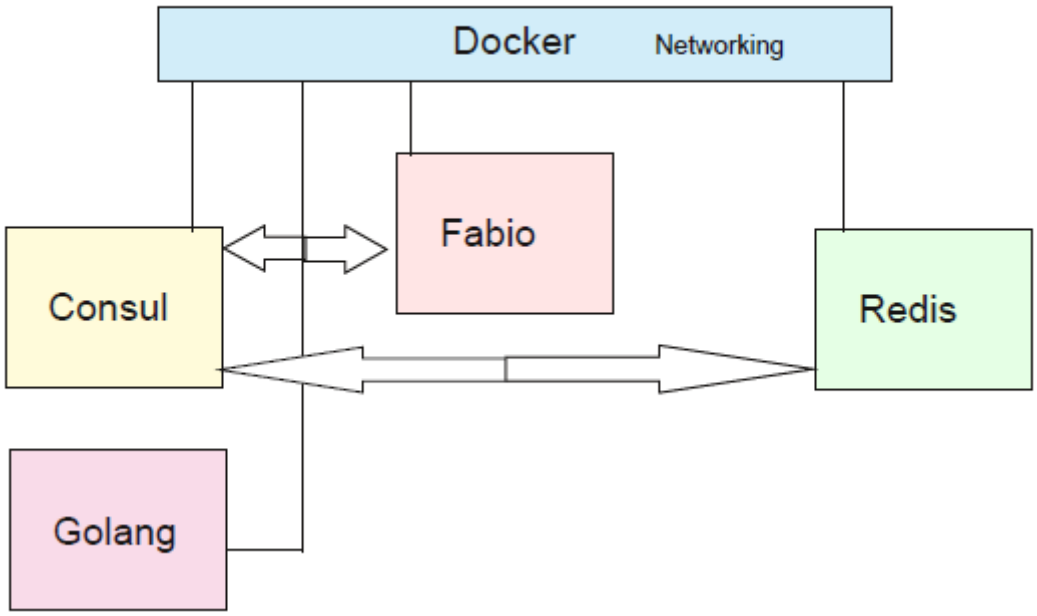
Result:

```
PS C:\Users\Yannick Arens\tmp\lab\fabio> curl -Method PUT http://localhost:8500/v1/agent/service/register -Body '{"name":"redis","address":"172.

StatusCode        : 200
StatusDescription : OK
Content           :
RawContent        : HTTP/1.1 200 OK
                    Content-Length: 0
                    Content-Type: text/plain; charset=utf-8
                    Date: Mon, 24 Oct 2016 18:02:15 GMT


Forms             : {}
Headers           : {[Content-Length, 0], [Content-Type, text/plain; charset=utf-8], [Date, Mon, 24 Oct 2016 18:02:15 GMT]}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 0
```

Check again the Consul interface and check if redis is now visible here.



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;
Unix:

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

Windows:

```
# mkdir 'c:/Users/<<Username>>/tmp/lab/golang'
```

Let's create a very simple test case and ensure everything is working as expected.

Unix:

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

Windows:

```
Use your own text editor to create main.go in c:\Users\<<Username>>\tmp\lab\golang\
```

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, let's start the Golang container, remember to do this in a new terminal window or tab;

Unix:

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

Windows:

```
# docker run -it --name golang -v '/c/Users/<<USERNAME>>/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.

-it	This is the main difference when we run the golang container.
-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 specificy 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.

```
PS C:\Users\Yannick Arens> docker run -it --name golang -v '/c/Users/Yannick Arens/tmp/lab/golang
root@8d9fed37247e:/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@b2c3e27e52fb:/tmp# ls
main.go
root@b2c3e27e52fb:/tmp#
```

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

```
root@b2c3e27e52fb:/tmp# go build main.go
root@b2c3e27e52fb:/tmp# ls
main  main.go
root@b2c3e27e52fb:/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.

Unix:

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

Windows:

```
Use your own text editor to create Dockerfile in c:\Users\<<Username>>\tmp\lab\golang\
```

First of all, we need to create a new file which will container 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.

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, navigate to the location where you have stored your Dockerfile. From there run the command below:

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

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

Unix:

```
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
```

Windows:

```
PS C:\Users\Yannick Arens\tmp\lab\golang> docker build -t yannickarens/golang1 .
Sending build context to Docker daemon 5.693 MB
Step 1 : FROM ubuntu
 ---> f7537d7788c5
Step 2 : ADD main /
 ---> a64519cfb08b
Removing intermediate container a5b3f246edbb
Step 3 : CMD /main
 ---> Running in fa9affff7940
 ---> a3aa9c82c992
Removing intermediate container fa9affff7940
Successfully built a3aa9c82c992
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and directories added to build conte
xes.
```

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
```

```
PS C:\Users\Yannick Arens\tmp\lab\golang> docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
yannickarens/golang1    latest              a3aa9c82c992        2 hours ago         132.9 MB
redis                   latest              190ed8a61620        2 days ago          182.9 MB
golang                  latest              f69c27b2f59a        3 days ago          672.4 MB
consul                  latest              2ba9010ee3cc        5 days ago          33.69 MB
ubuntu                  latest              f753707788c5        10 days ago         127.2 MB
magiconair/fabio        latest              c9492f8f0ea1        12 days ago         10.66 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 MicrosService way of doing it?

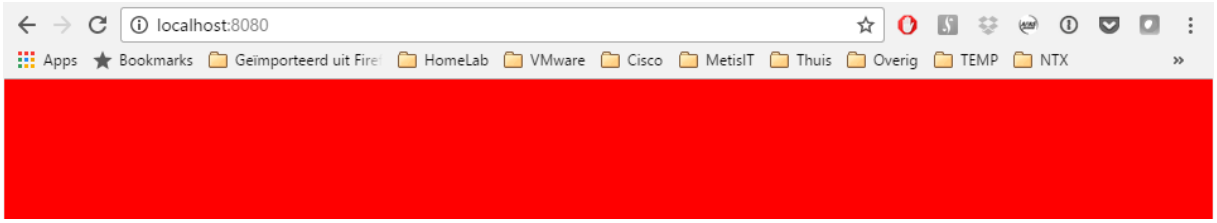
The last step in our testing is to run our new container. We do this in the same way as any other container.

```
# docker run -d -p 8080:8080 --name golang1 yannickarens/golang1
```

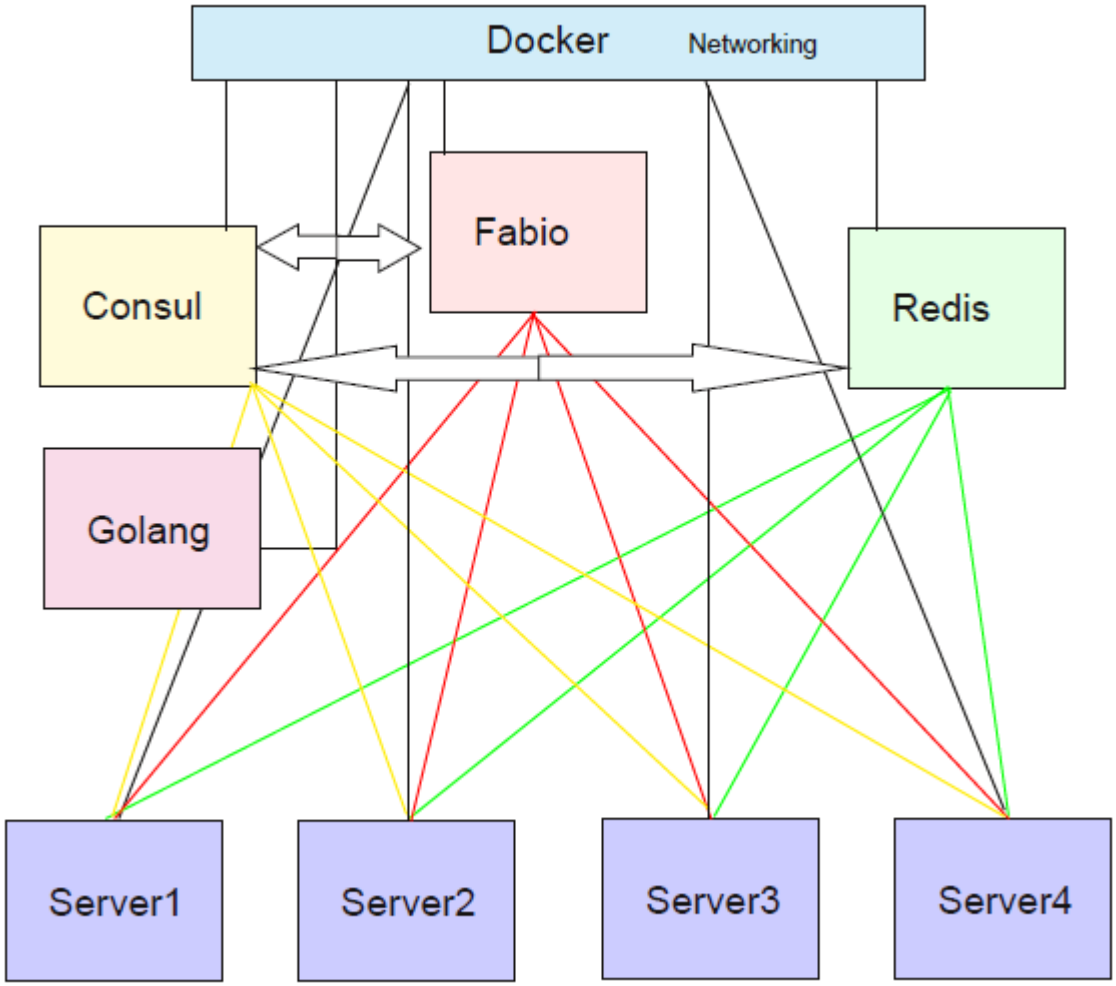
```
PS C:\Users\Yannick Arens\tmp\lab\golang> docker run -d -p 8080:8080 --name golang1 yannickarens/golang1
acbf42734d603e7ce90d22bae2025650fc8df2a28b457c95a152eb3a7bd6300b
```

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

And;



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 yannickarens/golang1
```

Next we need to change the go file we created before, it is probably easier to create some new

files and folders, so let's create four for now;

Unix:

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

Windows:

```
# mkdir c:\Users\<<Username>>\tmp\lab\golang\server1
# mkdir c:\Users\<<Username>>\tmp\lab\golang\server2
# mkdir c:\Users\<<Username>>\tmp\lab\golang\server3
# mkdir c:\Users\<<Username>>\tmp\lab\golang\server4
```

In each folder create a new Dockerfile;

Unix:

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

Windows

```
Use your own text editor to create Dockerfiles in
c:\Users\<<Username>>\tmp\lab\golang\serverX (see above)
```

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.

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. You need also to replace the <<IP>> field for the IP address of the console service in the main.go file.

Unix:

```
# 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
```

Windows:

```
Use your own text editor to create main.go files in each server folder
c:\Users\<<Username>>\tmp\lab\golang\serverX\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 webservers, 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.

```
package main

import (
   "fmt"
   "net/http"
   "log"
   "github.com/yannickarens/CloudNativeLab01/lab"
)

var redisServer string
var serverCount string
var dbStartCount string

func sayhello(w http.ResponseWriter, r *http.Request) {
   color := "red"
   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 := "172.17.0.2"
   portWeListenOn := "8080"

   result, err := lab.RegisterMe(ipOfConsul, urlsToRegister, portWeListenOn )
   redisServer = lab.GetServiceAddress("redis")
   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 {
         log.Fatal("ListenAndServe: ", err)
      }
   } else {
      fmt.Println(err)
   }
}
```

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/yannickarens/CloudNativeLab01/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 Rob Porter wrote specifically for the Cisco Cloud PVT.

```
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 := "red"
   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 webservers 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.

```
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 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 := "172.17.0.2"
   portWeListenOn := "8080"

   result, err := lab.RegisterMe(ipOfConsul, urlsToRegister, portWeListenOn )
   redisServer = lab.GetServiceAddress("redis")
   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 {
         log.Fatal("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.

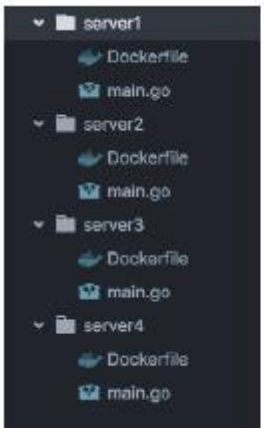
The next three lines are some variables we need for registration into Consul. The first one is a simple a list of the URL's that our servers care about, in this case it is only /. The other two are hopefully self-explanatory.

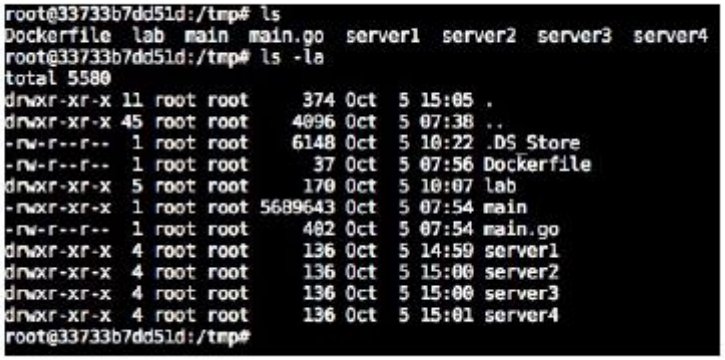
The next four lines are getting some data from the custom package built for the lab, the first registers our server in Consul, the second gets the number of servers registered in Consul and the third gets some data from our redis server.

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

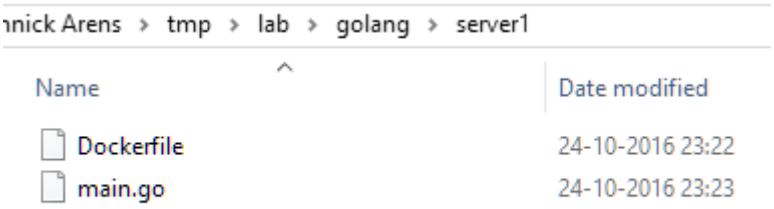
You should now have a structure very similar to the following.

Unix:





Windows:



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/yannickarens/CloudNativeLab01/lab
```

This can take some time.

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@b2c3e27e52fb:/tmp/server1# CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
root@b2c3e27e52fb:/tmp/server1# cd ..
root@b2c3e27e52fb:/tmp# cd server2
root@b2c3e27e52fb:/tmp/server2# CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
root@b2c3e27e52fb:/tmp/server2# cd ..
root@b2c3e27e52fb:/tmp# cd server3
root@b2c3e27e52fb:/tmp/server3# CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
root@b2c3e27e52fb:/tmp/server3# cd ..
root@b2c3e27e52fb:/tmp# cd server4
root@b2c3e27e52fb:/tmp/server4# CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
```

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.

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

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

```
PS C:\Users\Yannick Arens\tmp\lab\golang> docker build -t yannickarens/server3 server3/
Sending build context to Docker daemon 6.434 MB
Step 1 : FROM scratch
 --->
Step 2 : ADD main /
 ---> d9a6ad510244
Removing intermediate container ce8e367351a4
Step 3 : CMD /main
 ---> Running in 2eeca46213e7
 ---> 43bf04f4516d
Removing intermediate container 2eeca46213e7
Step 4 : EXPOSE 8080
 ---> Running in 26e742012373
 ---> 4ae39f0d926e
Removing intermediate container 26e742012373
Successfully built 4ae39f0d926e
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host.
```

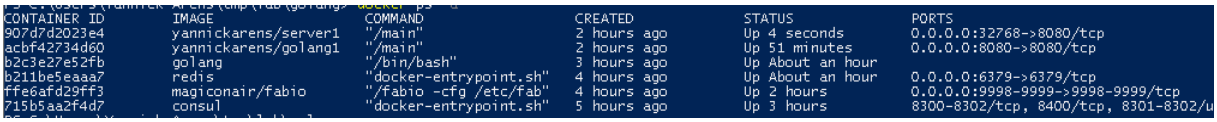
If all has gone well, when you now list your docker images, you will see there are now four new images and they are now very small roughly 6.5MB in size!

```
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
yannickarens/server4    latest              de38cee7a007        2 hours ago         6.429 MB
yannickarens/server3    latest              4ae39f0d926e        2 hours ago         6.429 MB
yannickarens/server2    latest              d84696af0baa        2 hours ago         6.429 MB
yannickarens/server1    latest              0e42fff75f78        2 hours ago         6.429 MB
yannickarens/golang1    latest              a3aa9c82c992        2 hours ago         132.9 MB
redis                   latest              190ed8a61620        2 days ago          182.9 MB
golang                  latest              f69c27b2f59a        3 days ago          672.4 MB
consul                  latest              2ba9010ee3cc        5 days ago          33.69 MB
ubuntu                  latest              f753707788c5        11 days ago         127.2 MB
magiconair/fabio        latest              c9492f8f0ea1        12 days ago         10.66 MB
```

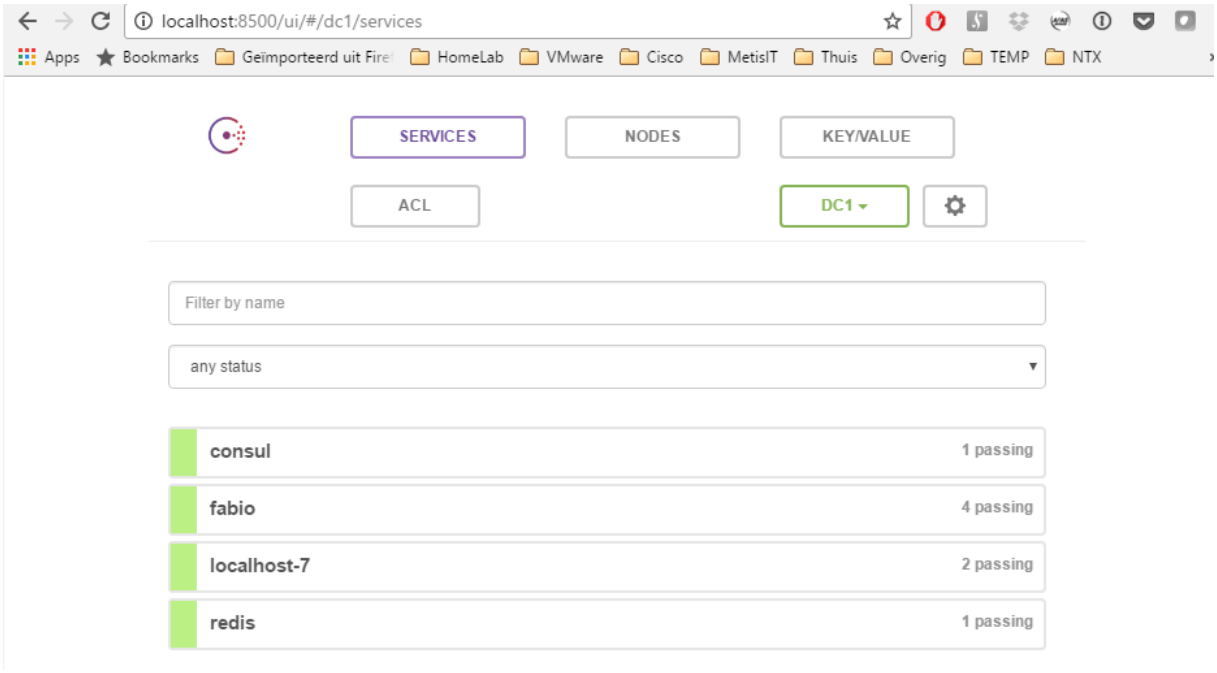
Finally we need to start our servers, let's try one first

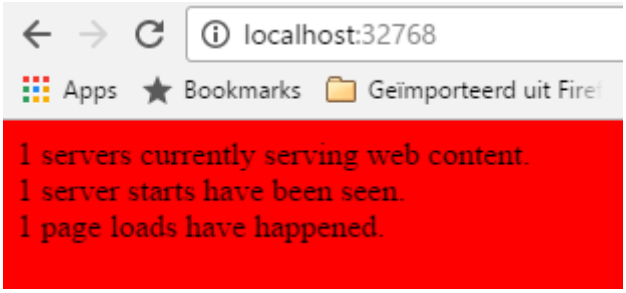
```
# docker run -d -P --name server1 yannickarens/server1
```

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;



```
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS              PORTS
907d7d2023e4        yannickarens/server1   "/main"                  2 hours ago         Up 4 seconds        0.0.0.0:32768->8080/tcp
acbf42734d60        yannickarens/golang1   "/main"                  2 hours ago         Up 51 minutes       0.0.0.0:8080->8080/tcp
b2c3e27e52fb        golang                 "/bin/bash"              3 hours ago         Up About an hour
b211be5eaaa7        redis                  "docker-entrypoint.sh"   4 hours ago         Up About an hour    0.0.0.0:6379->6379/tcp
ffe6afd29ff3        magiconair/fabio       "/fabio -cfg /etc/fab"   4 hours ago         Up 2 hours          0.0.0.0:9998-9999->9998-9999/tcp
715b5aa2f4d7        consul                 "docker-entrypoint.sh"   5 hours ago         Up 3 hours          8300-8302/tcp, 8400/tcp, 8301-8302/u
```

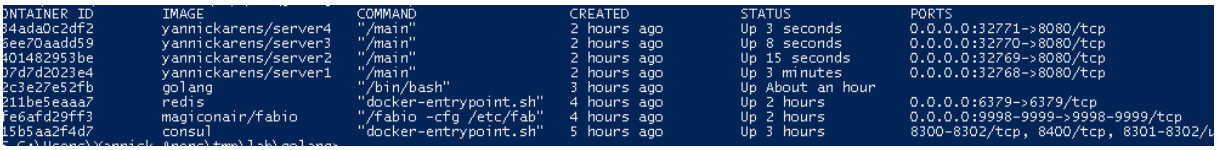


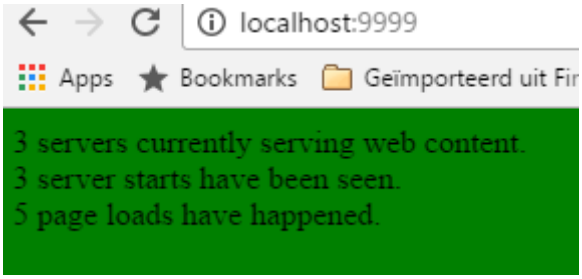


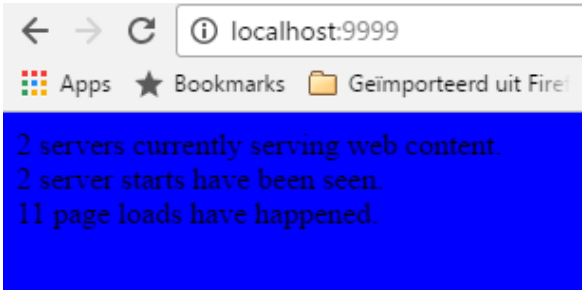
Now start the remaining servers;

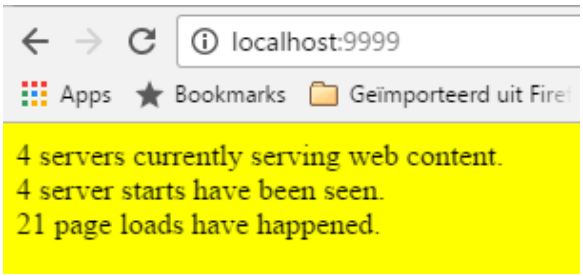
```
# docker run -d -P --name server2 yannickarens/server2
# docker run -d -P --name server3 yannickarens/server3
# docker run -d -P --name server4 yannickarens/server4
```

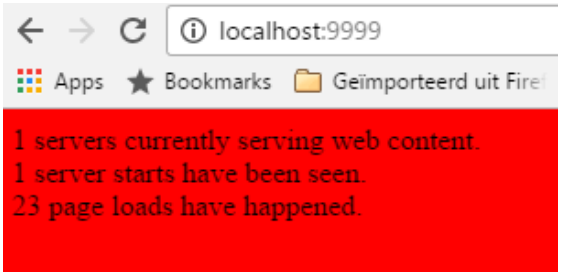
And check everything has worked as expected. If you now go to http://localhost:9999 then the load balancer should take you to one of our 4 webservers











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.