Blue-Green Deployment

```
$ git clone
https://github.com/foundjem/bluegreen-eployment.git

$ cd bluegreen-eployment
```

Initialize the swarm cluster

Make sure that your docker instance is in *Swarm-mode*. For a single node, the command is simply the following:

```
$ docker swarm init

------
Swarm initialized: current node (2wr3rr195xt1bvr7cuwrw3mp8) is now a manager.

To add a worker to this swarm, run the following command:
    docker swarm join \
    --token
SWMTKN-1-343oklx5y9zqwc129p4ttvgxeeexh49vpgslavcnqzipjmp2n4-0375hm0z4nto02l8jbt08ga
1n \
    192.168.65.2:2377

To add a manager to this swarm, run the following command:
    docker swarm join \
    --token
SWMTKN-1-343oklx5y9zqwc129p4ttvgxeeexh49vpgslavcnqzipjmp2n4-bnfuijeolp487mllvj7l9sj
il \
    192.168.65.2:2377
```

Creating the Network

We'll need a backend network for our edge and backend services to communicate on. This traffic will run on an overlay network and only be accessible to other services attached to the same network.

```
$ docker network create --driver overlay backend

-----
6u9nez4oadt63nadfxgbvwmss
```

Deploy our Service

We are now ready to deploy the first version of our service. First, we build the image `myapp` from the directory `./myapp`. `myapp` is a simple web service that displays the container's hostname. It also displays the contents of an http header `Color` or `unknown`, if the header is not set.

All commands are done from the sillyproxy root.

Let's start by building the image specified by the **Dockerfile**:

```
$ docker build -t myapp myapp/

---------
Sending build context to Docker daemon 3.584 kB
Step 1 : FROM golang:1.7-alpine
 ---> 52493611af1e
Step 2 : COPY . /go/src/github.com/stevvooe/sillyproxy/myapp
 ---> 489c65a768e7
Removing intermediate container 3195d1e76f24
Step 3 : RUN go install github.com/stevvooe/sillyproxy/myapp
 ---> Running in 08b2e8a0bef0
 ---> edf64a6a1894
Removing intermediate container 08b2e8a0bef0
Step 4 : EXPOSE 8080
 ---> Running in 47c58faeb679
 ---> 3f83e0bf1717
Removing intermediate container 47c58faeb679
Step 5 : ENTRYPOINT /go/bin/myapp
 ---> Running in 77b163b18105
 ---> 34436cbe5b50
Removing intermediate container 77b163b18105
Successfully built 34436cbe5b50
```

Once the image is built, we are ready to run our first service. We are going to bind it to the external port 7999 for canary testing, but that is not necessary. We also attach it to the backend network so we can use it with our proxy later.

```
$ docker service create --name myapp-v1 --network backend -p7999:8080 myapp
```

```
---------
ekfaa2wfbxj2j22l6gay0iqoa
```

We can use `docker service ls` to confirm it is running:

```
$ docker service ls

---------
ID            NAME      REPLICAS  IMAGE  COMMAND
ekfaa2wfbxj2  myapp-v1  1/1       myapp
```

We can also see that we get the expected result by curling the service endpoint:

```
$ curl http://localhost:7999

---------
unknown e99a62b52d1d
```

Using the Proxy

While the above could be used to scale and update the service, we'd like to have a little more control over our traffic. Specifically, we'd like to be able to serve up different running versions of the service and direct different amounts of traffic to make sure we are good.

Again, from the project root, let's build an image for our proxy.

```
$ docker build -t sillyproxy .

-----------
Sending build context to Docker daemon 480.3 kB
Step 1 : FROM golang:1.7-alpine
 ---> 52493611af1e
Step 2 : COPY . /go/src/github.com/stevvooe/sillyproxy
 ---> f732e09a9d28
Removing intermediate container 57b0d3368d4a
Step 3 : RUN go install github.com/stevvooe/sillyproxy
 ---> Running in 2cac4703e808
 ---> 400ed84a617a
Removing intermediate container 2cac4703e808
Step 4 : ENTRYPOINT /go/bin/sillyproxy
 ---> Running in a2194801b359
 ---> c42d98b3b82c
```

```
Removing intermediate container a2194801b359
Successfully built c42d98b3b82c
```

With that image, we will create our proxy, directed towards our service:

```
$ docker service create --name proxy --network backend -p8080:8080

-----------
-eBLUE=http://myapp-v1:8080 sillyproxy
6e4pqdozk8wh7s3zou1af7g1r
```

It's important to note that we've exposed the service on port 8080 across the cluster *and* attached the proxy to the backend network. Anything attached to the same backend can be accessed using the name of the service as a DNS value. Since `myapp-v1` is also on backend, we just use the service name to configure the URL to use for the `BLUE`service.

We can confirm this is working with curl:

```
$ curl -v localhost:8080/

---------
blue e99a62b52d1d
```

Notice that this is the same container id from the backend, which is available on port 7999, except that the proxy has set the `Color` header:

```
$ curl localhost:7999

--------
master ✱
unknown e99a62b52d1d
```

At this point, we could remove the export of 7999 if no longer need to get to the service directly.

Scaling the Service

At this point, we are looking great for production, except that we are running a single instance. If an instance goes down or we need to spread load across a set of nodes, we can add more instances to the backend. We do this by setting the number of *replicas* for a service.

To see the current number of replicas, we use the `docker service ls` command:

```
$ docker service ls

---------
ID            NAME      REPLICAS  IMAGE       COMMAND
6e4pqdozk8wh  proxy     1/1       sillyproxy
ekfaa2wfbxj2  myapp-v1  1/1       myapp
```

For our use case, we'll need two proxy instances and four backends. We do this using the `docker service scale`command:

```
$ docker service scale myapp-v1=4 proxy=2
```

After the new containers are started, you'll see the following:

```
$ docker service ls

---------
ID            NAME      REPLICAS  IMAGE       COMMAND
6e4pqdozk8wh  proxy     2/2       sillyproxy
ekfaa2wfbxj2  myapp-v1  4/4       myapp
```

Using curl to hit the proxy, we can see we get four different backends:

```
$ curl localhost:8080
```

```
blue 7f5d3eacb82d
```

```
$ curl localhost:8080
```

```
blue 7f5d3eacb82d
```

```
$ curl localhost:8080
```

```
blue 788e2a92dbce
```

```
$ curl localhost:8080
```

```
blue 788e2a92dbce
```

```
$ curl localhost:8080
```

```
blue 7f973b247aff
```

```
$ curl localhost:8080
```

```
blue e99a62b52d1d
```

By using the service name `myapp-v1`, that we configured when we created the proxy service, docker will route connections to all the backends available as the service scales. This uses a linux kernel feature called IPVS and a gossip network to notify peers of the locations for running replicas of a service. All we've told the proxy to do is hit `http://localhost:8080` and docker is doing the rest. Even though we are hitting `localhost:8080` for the proxy, those connections are also being load balanced between the two instances of the proxy.

The same result can be achieved by hitting the service directly on port 7999, expect that the `Color` value will be unknown.

Deploying a new version

At this point, we'd like to deploy a new version of our service. `myapp` has a feature switch that will display the container id in HTML rather than plain text. This is activated by the environment variable `V2` but this could just as well be another image.

Let's create the new service and confirm it is running:

```
$ docker service create --name myapp-v2 --network backend -e V2=1 -p7998:8080 myapp

-----------
f5l4xujsn904uhu2xv229rgez
$ docker service ls
ID            NAME      REPLICAS  IMAGE       COMMAND
6e4pqdozk8wh  proxy     2/2       sillyproxy
ekfaa2wfbxj2  myapp-v1  4/4       myapp
f5l4xujsn904  myapp-v2  1/1       myapp
```

We've made the new service accessible on port 7998 for testing. Let's go ahead and hit with `curl`:

```
$ curl localhost:7998
master ✱

<h1>be0dc0efaba2 (unknown)</h1>
```

Woohoo! HTML!!!

Clearly, production is ready. Let's add this to `sillyproxy`, but let's be conservative and only route 20% of traffic to the new version. We do this using the `docker service update` command with additional environment variables:

```
$ docker service update --env-add GREEN=http://myapp-v2:8080/ --env-add
GREEN_WEIGHT=1 --env-add BLUE_WEIGHT=4 proxy
```

We add `GREEN`, with a weight of 1, and modify `BLUE` to have a weight of 4. The above kicks off a rolling update of the service with new environment variables.

Any proxy can be setup to coordinate this setup using their own weight system.

With `curl`, we can see that certain requests receive HTML, rather than plain text:

```
$ curl localhost:8080
```

```
blue e99a62b52d1d
```

```
$ curl localhost:8080
```

```
blue 7f5d3eacb82d
```

```
$ curl localhost:8080
```

```
blue 788e2a92dbce
```

```
$ curl localhost:8080
```

```
<h1>be0dc0efaba2 (green)</h1>
```

```
$ curl localhost:8080
```

```
blue 7f973b247aff
```

```
$ curl localhost:8080
```

```
<h1>be0dc0efaba2 (green)</h1>
```

```
$ curl localhost:8080
```

```
blue e99a62b52d1d
```

Moving to Green

People are wild about HTML! UX testing has confirmed and we are ready to go `GREEN` across the board.

Before we do this, let's scale the `myapp-v2`, the `GREEN` backend, to production levels:

```
$ docker service scale myapp-v2=4

--------
myapp-v2 scaled to 4
```

Now, that we are ready, we just kick off another rolling update to go full green:

```
$ docker service update --env-rm GREEN_WEIGHT --env-rm BLUE_WEIGHT --env-rm BLUE
proxy

-------
proxy
```

Let's use `docker service ls` to monitor the deployment:

```
$ docker service ls
ID            NAME      REPLICAS  IMAGE       COMMAND
6e4pqdozk8wh  proxy     2/2       sillyproxy
ekfaa2wfbxj2  myapp-v1  4/4       myapp
f514xujsn904  myapp-v2  4/4       myapp
```

Hitting the endpoint, we can see that we now only hit the `GREEN` backend, with the HTML output, load-balanced among the `myapp-v2` replicas:

```
$ curl localhost:8080
```

`<h1>583ef5bfc936 (green)</h1>`

```
$ curl localhost:8080
```

`<h1>be5901cf1337 (green)</h1>`

```
$ curl localhost:8080
```

`<h1>bb907389b5ea (green)</h1>`

Rollback

Well, it turns out that HTML is costing a massive amount of bandwidth and we cannot it afford the extra traffic. We have to rollback.

Luckily, this is just another rolling update:

```
$ docker service update --env-rm GREEN --env-add BLUE=http://myapp-v1:8080 proxy

------
```

```
proxy
```

And we are back to plain text:

```
$ curl localhost:8080

------
blue e99a62b52d1d
```