Docker and the Path to a Better Staging Environment

...

Gil Tayar (@giltayar)
March 2019
This presentation: http://bit.ly/docker-and-the-path-devopspro
https://github.com/giltayar/docker-and-the-path

About Me

- My developer experience goes all the way back to the ‘80s.
- Am, was, and always will be a developer
- Testing the code I write is my passion
- Currently evangelist and architect @ Applitools
- We deliver Visual Testing tools:
  If you’re serious about testing, checkout Applitools Eyes

- Sometimes my arms bend back
- But the gum I like is coming back in style

Staging Environments

Staging Environments

- The environment where the application is deployed to, prior to deployment in production
- Also used prior to integrating code between teams
- Usually one environment, but can be more
- And, unfortunately...

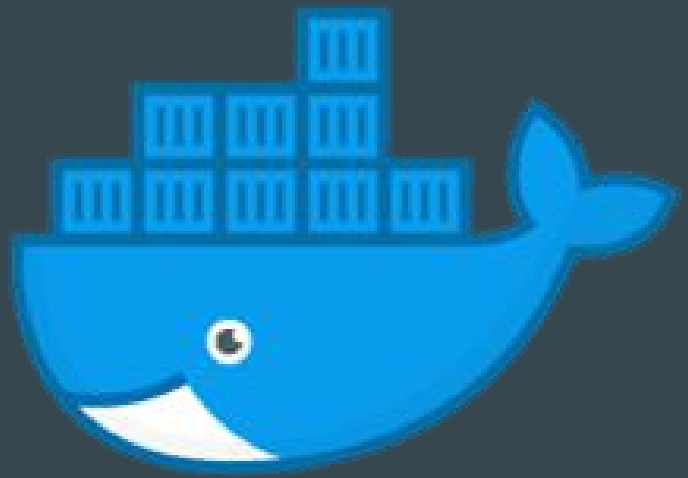
Almost, but not quite, entirely unlike production

(apologies to Douglas Adams)

Staging Environments

- Built using a combination of chewing gum and baling wire
- Not maintained as well as the production environment
- Not the same deployment procedure
- Not the same production infrastructure

And this is where we test...



docker

Revolution!

Revolution in...

- How apps are developed
- How apps are tested
- How apps are deployed to production

This Talk

- What Docker is
- How to use it with Docker-compose
- How to use it with Kubernetes
- How to build a production and staging environment with it

Put your seatbelt on, it’s going to be a wild and technical ride

End Goal

- Run our application in a staging environment using Docker
- Simple application:
  - MongoDB database
  - Blogging web application (frontend and backend)

How would we run Mongo in a staging environment?

- Install it on one of the machines in Staging

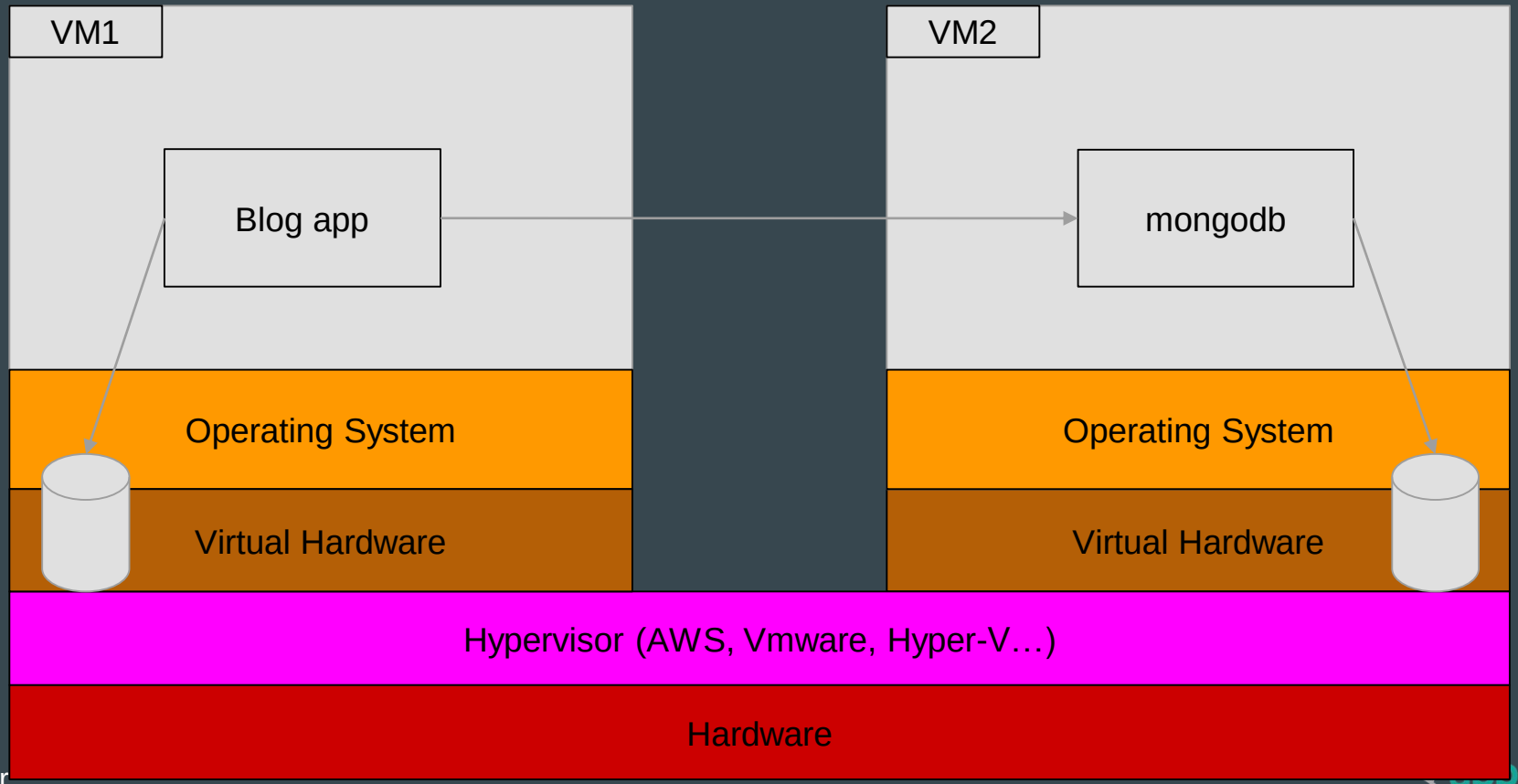
or...

- Build a VM image that includes Mongodb, and run it on AWS, Azure, VMware...

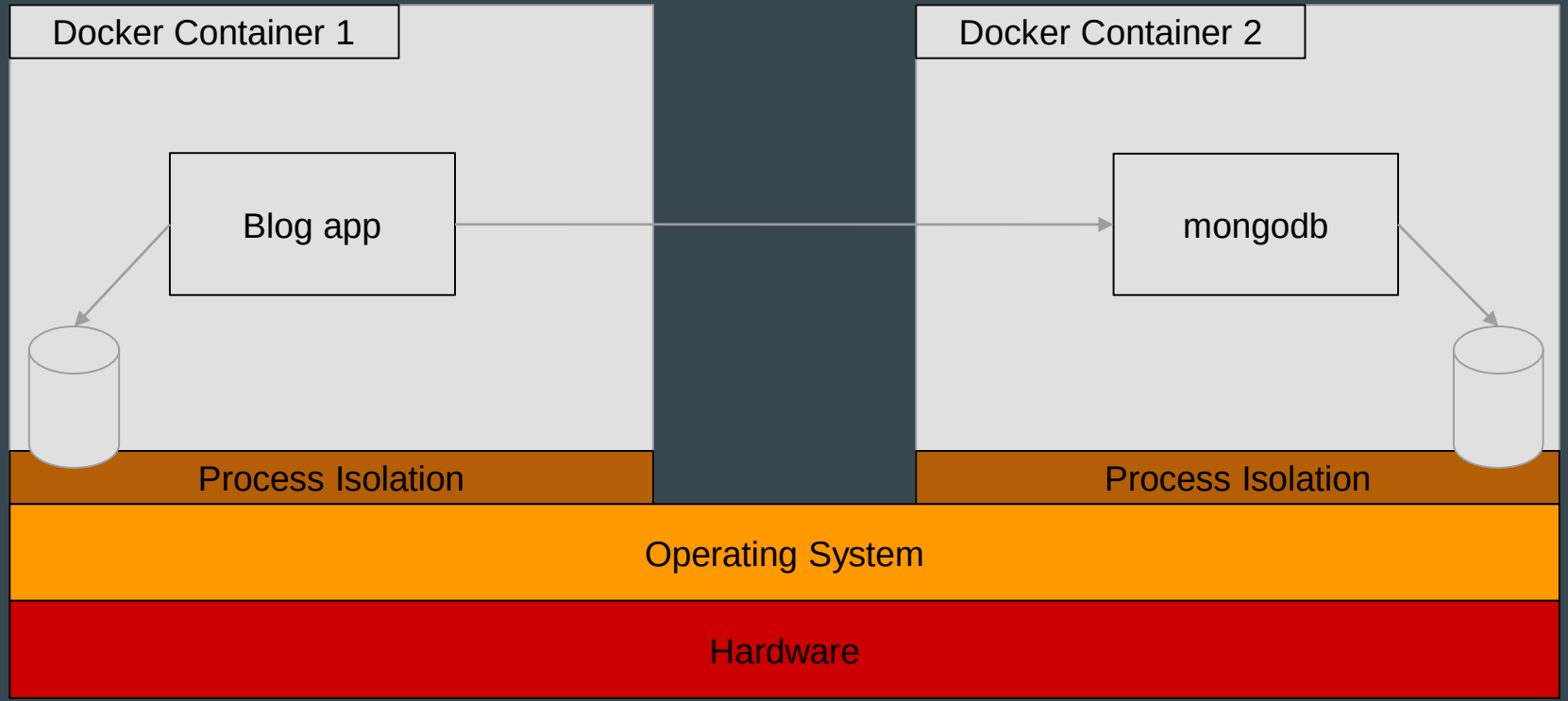
Problematic

- Installing:
  - Automated install scripts are not easy
- VM image:
  - We can run two!
  - Not easy building the image
  - Every version, the VM needs to be rebuilt
  - Start time is minutes.
  - But it’s the better option.

The VM Option



The Docker Option



Let’s Run MongoDB under Docker

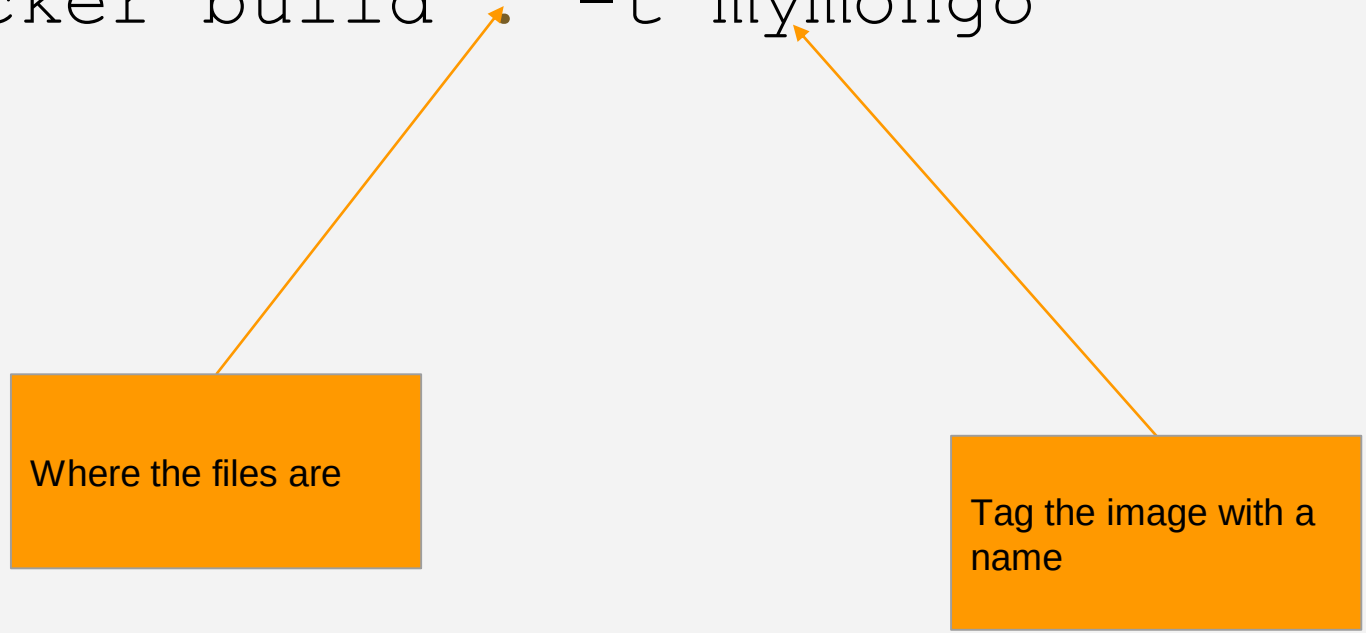
Dockerfile

```
FROM ubuntu

WORKDIR /data
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update -y
RUN apt-get install -q -y mongodb

RUN mkdir -p /data/db

EXPOSE 27017

CMD ["mongod"]
```

- An image is a frozen container
- A template for docker containers
- Set of instructions that is run in the container:
  - Runs the base image (FROM)
  - Executes the RUN-s (using WORKDIR and ENV)
  - Adds some metadata to container (CMD, EXPOSE)
  - Freezes the result into a docker image

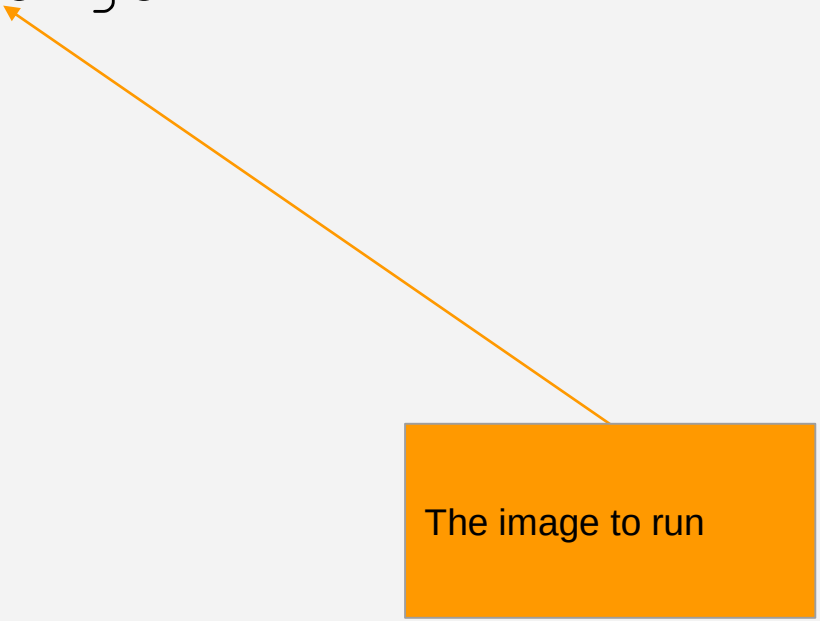
Building the docker image

```
docker build . -t mymongo
```

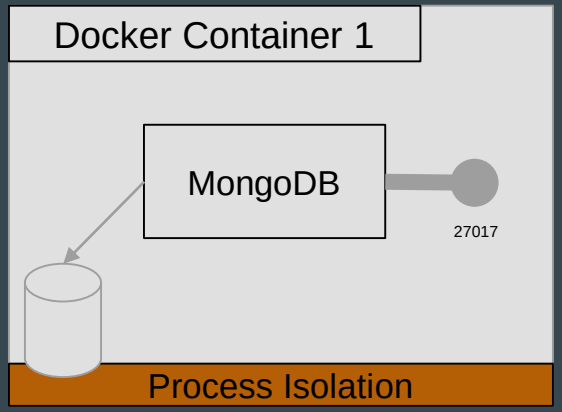


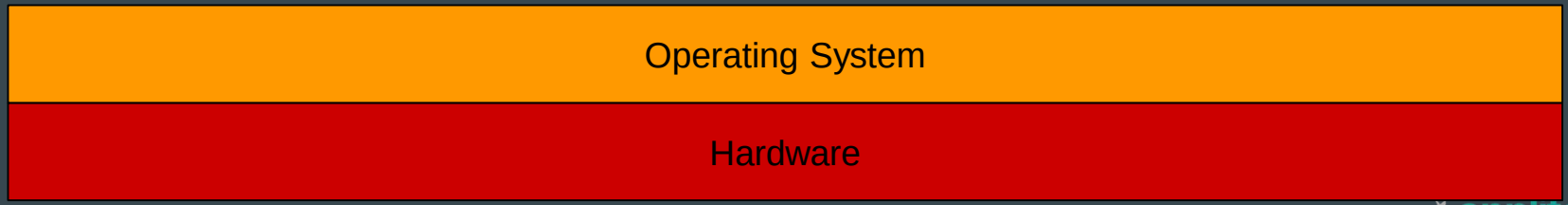
Running a container

```
$ docker run mymongo
```



Yay! It's working

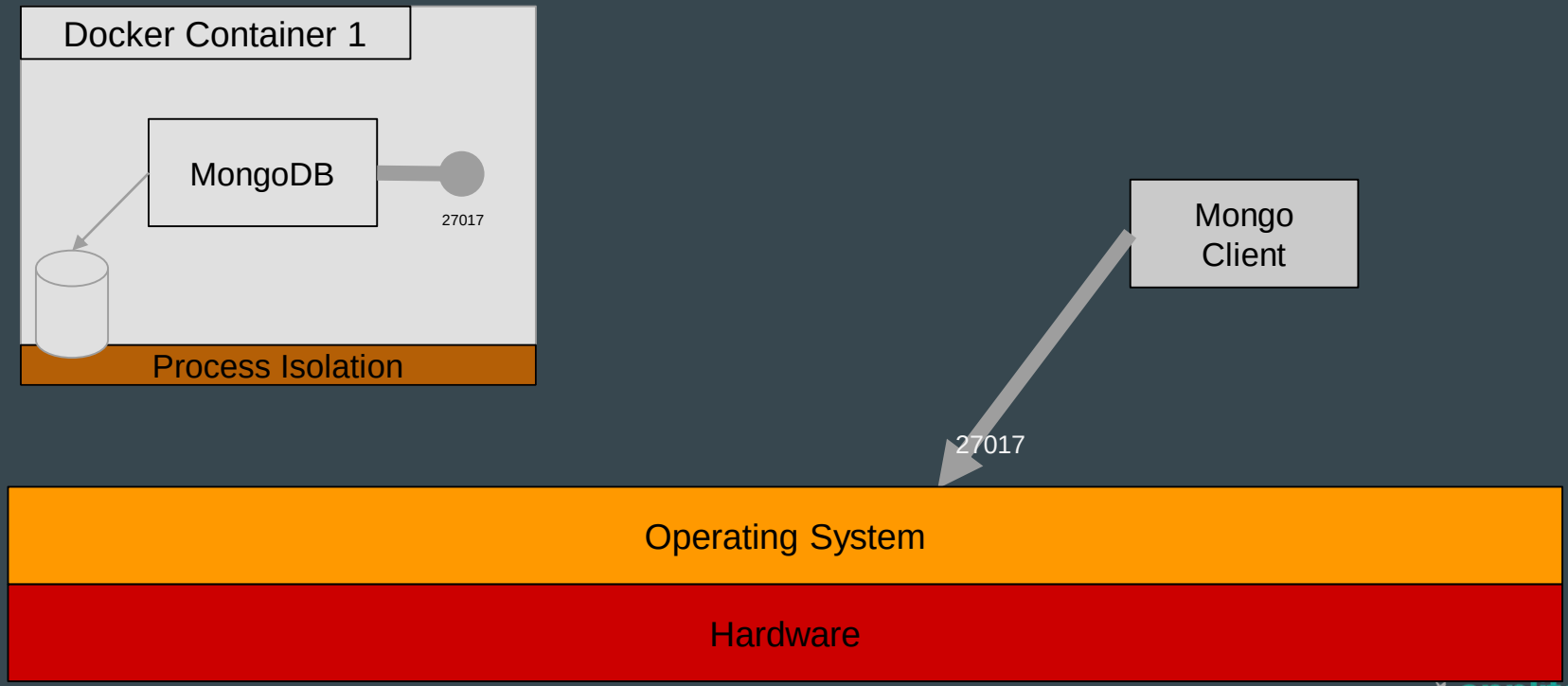




Connecting to mongo via mongo client

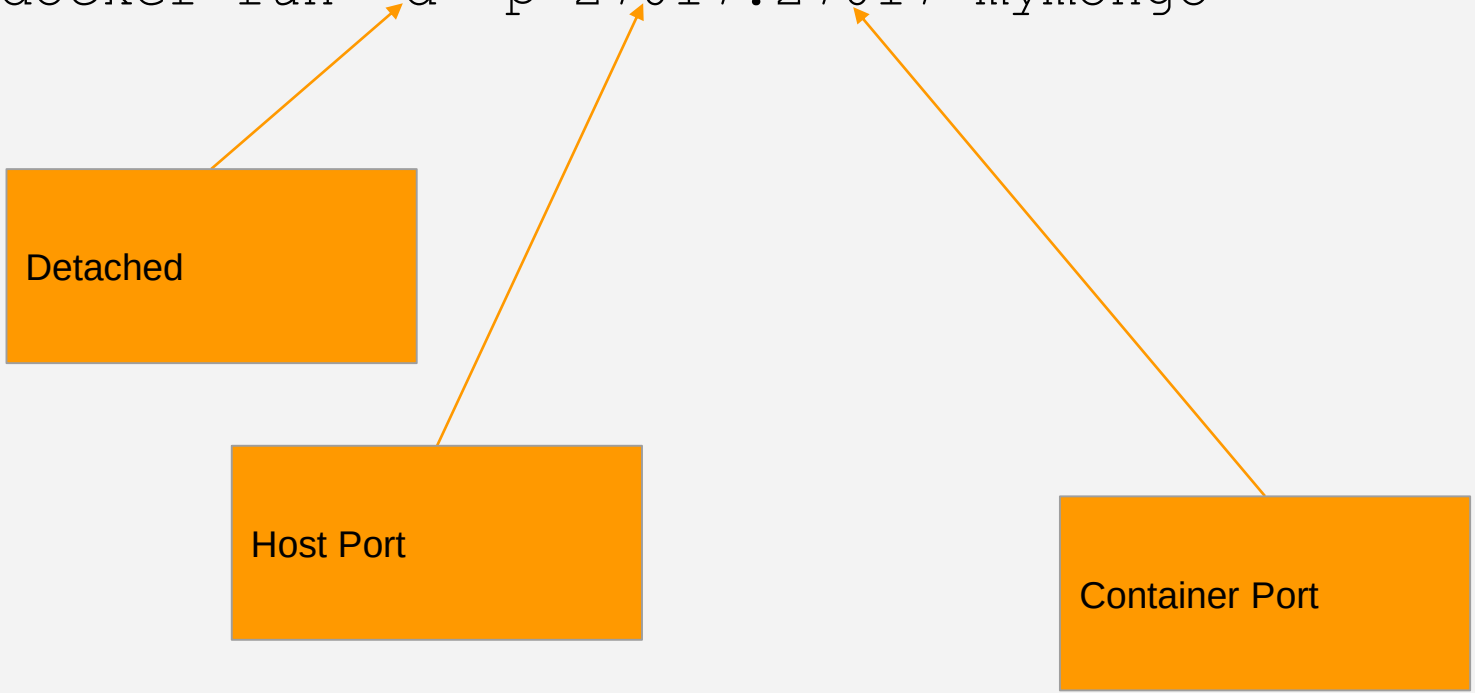
```
$ mongo
```

Why isn't it connecting? (Docker and TCP Ports)

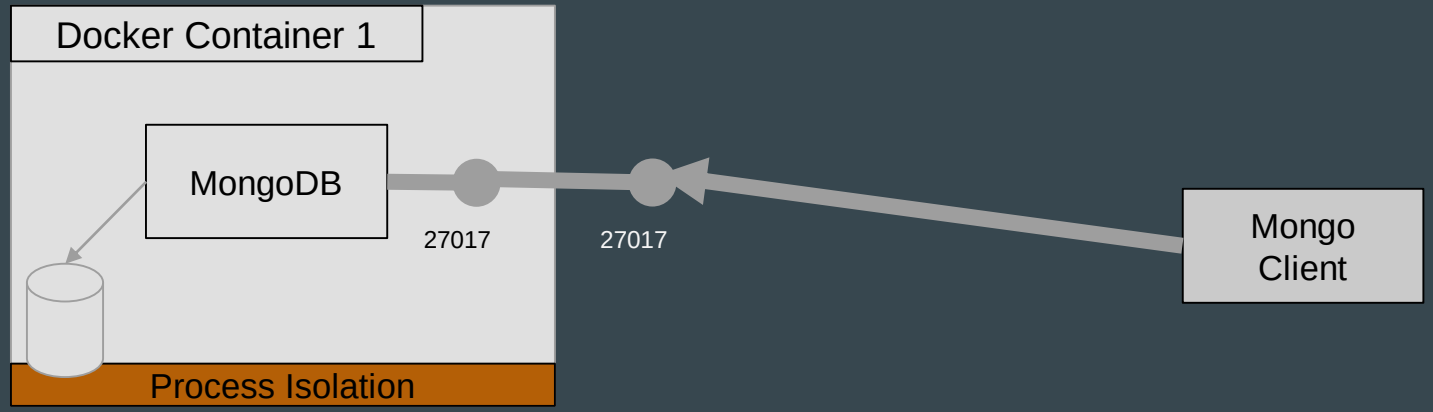


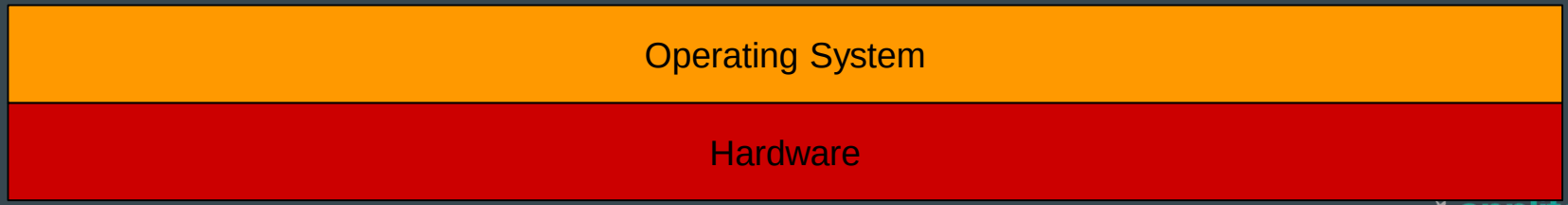
Running a container with port mapping

```
docker run -d -p 27017:27017 mymongo
```



This is what port mapping looks like

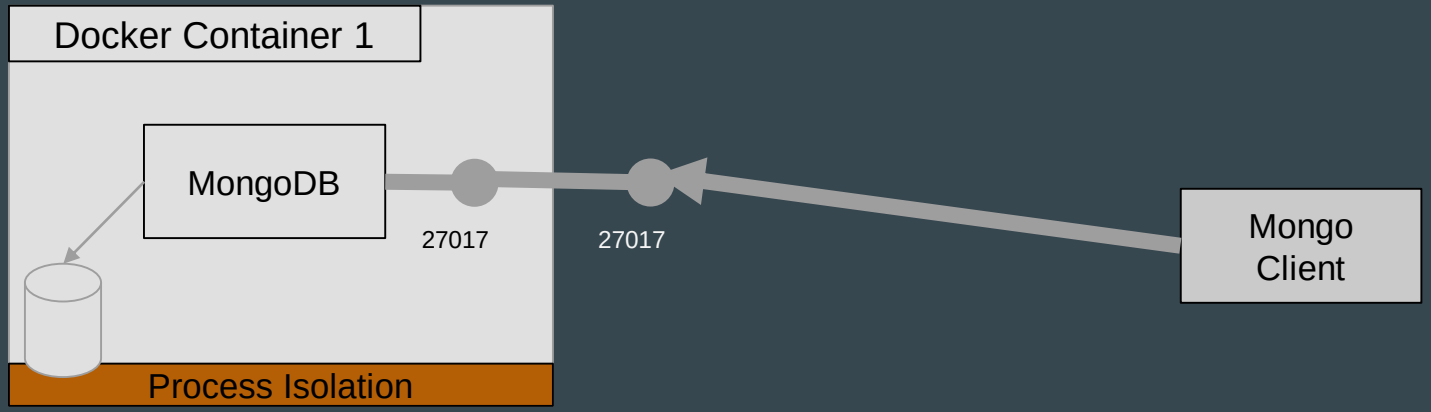


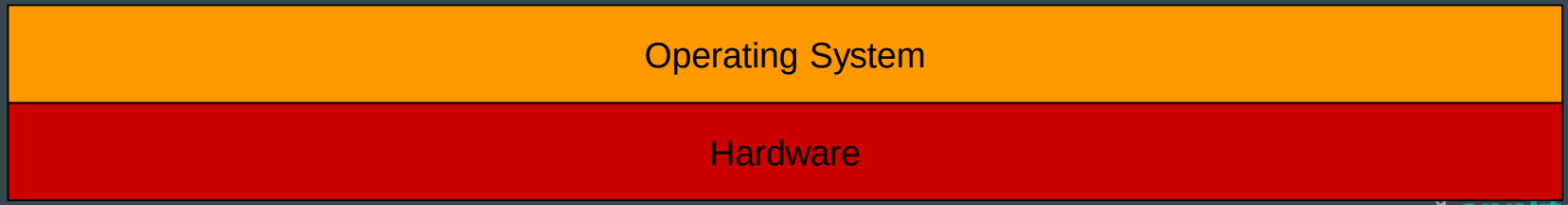


Connecting with Mongo Client

```
$ mongo
> db.users.save({username: 'giltayar', name: 'Gil Tayar'})
> db.users.find()
{ "_id" : ObjectId("5abf3d2bdef8a36d505d3732"), "a" : 1 }
```

And... it's connecting!





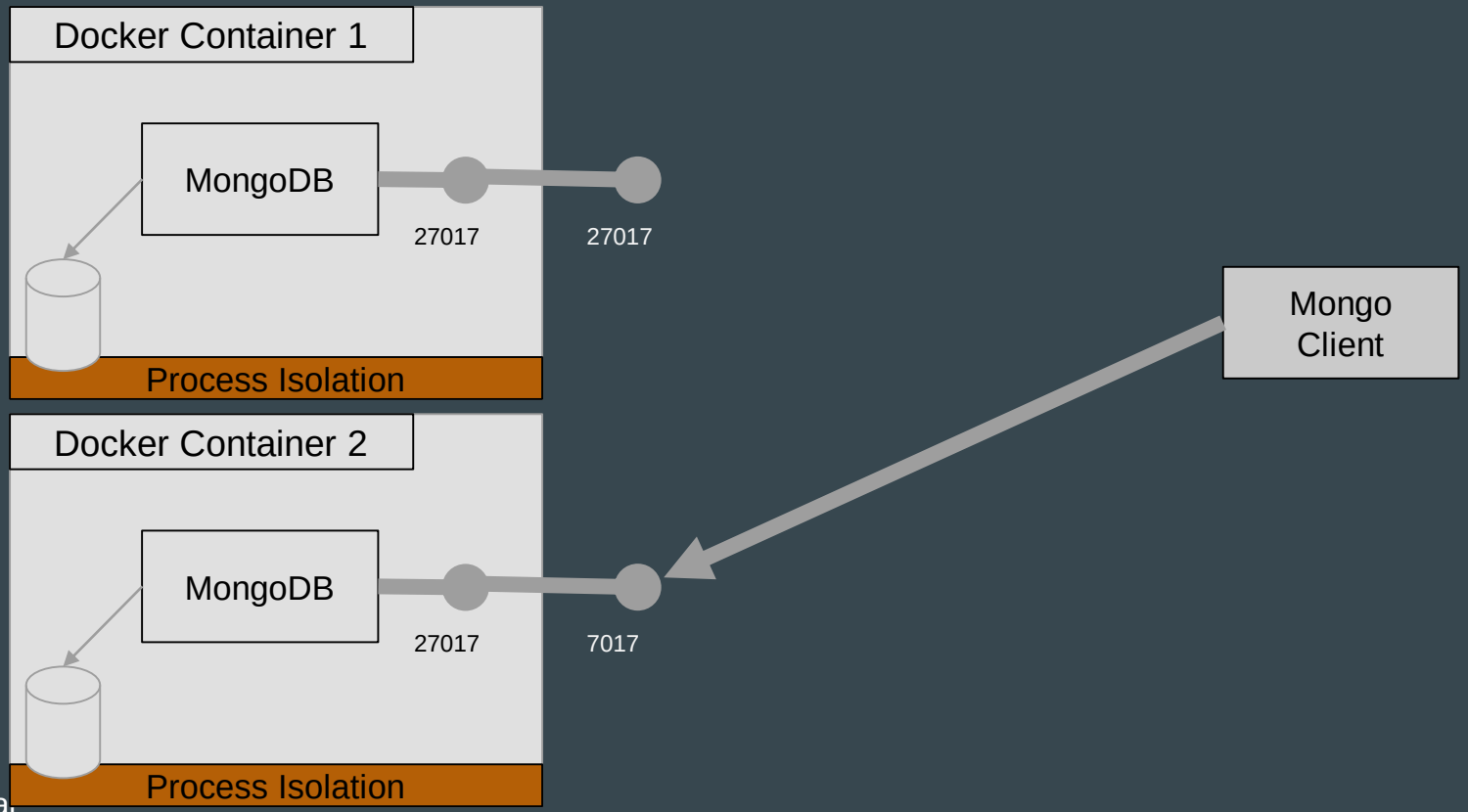
Connecting with Mongo Client

```
$ mongo
> db.users.save({username: 'giltayar', name: 'Gil Tayar'})
> db.users.find()
{ "_id" : ObjectId("5abf3d2bdef8a36d505d3732"), "a" : 1 }
```

Running another Mongo container

```
$ docker run -d -p 7017:27017 mymongo
$ mongo localhost:7017
> db.users.save({username: 'giltayar', name: 'Gil Tayar'})
> db.users.find()
{ "_id" : ObjectId("5abf3d2bdef8a36d505d3732"), "a" : 1 }
```

Two containers, different host ports



Maintenance of Docker Containers

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS              PORTS
6e33d46a915c        mymongo             "mongod"            Less than a second ago   Up 1 second         0.0.0.0:27017->27017/tcp
44ce252702e6        mymongo             "mongod"            9 seconds ago            Up 11 seconds       0.0.0.0:7017->27017/tcp
```

```
$ docker rm 6e33d46a915c
```

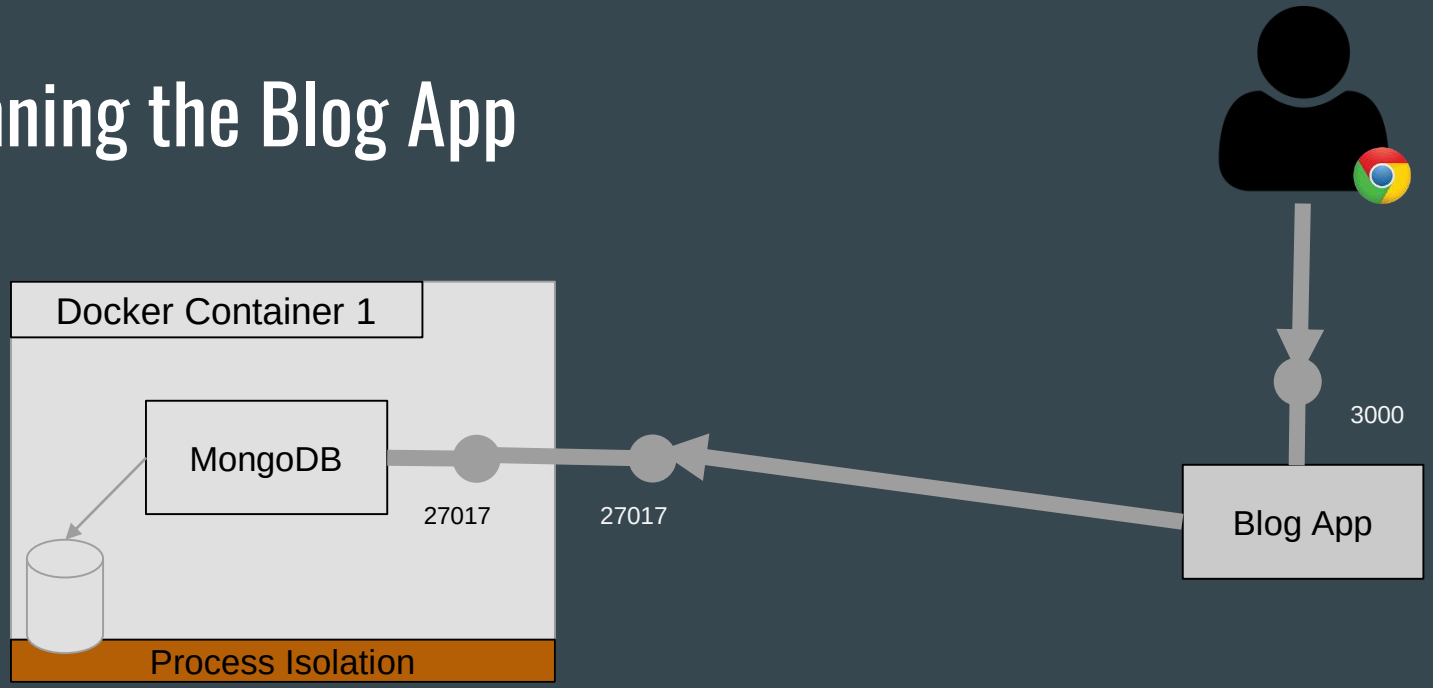
```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
44ce252702e6        mymongo             "mongod"            9 seconds ago       Up 11 seconds       0.0.0.0:7017->27017/tcp
```

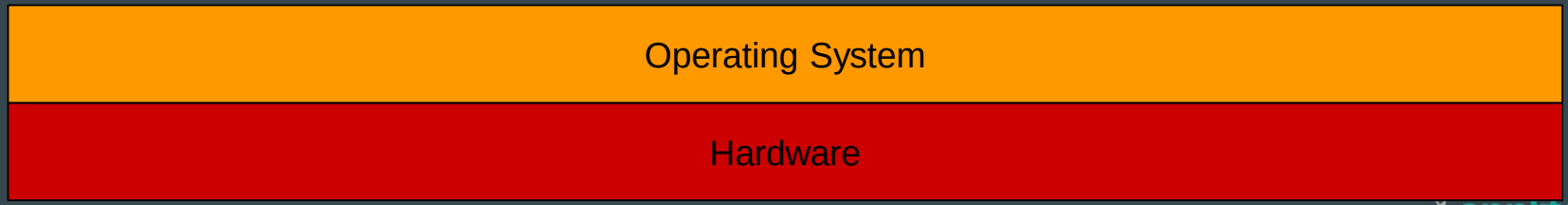
Let’s Run The Blog App

Running the Blog App Locally

```
$ cd ../blog-app-example
$ npm install
$ npm run build:frontend
$ npm run dev
```

Running the Blog App





Dockerfile

```
FROM node

WORKDIR app
ENV NODE_ENV=production

COPY . .
RUN npm install --production

EXPOSE 3000

CMD ["node", "src/app.js"]
```

- “node” is an image that already has NodeJS installed
  - Thousands of these pre-built images in hub.docker.com.
- COPY copies from host directory into docker directory

Building the Blog App

```
$ cd ../blog-app-example

$ npm run build:frontend

$ docker build . -t myblog
```

Running the App

```
$ docker run -t -p 3000:3000 giltayar/blog-app-example

...
Error: secret should be set
    at module.exports (.../index.js:21:42)
    at Object.<anonymous> (/app/src/routes/auth.js:14:13)
```

Twelve Factor Apps

- A set of rules on how to run modern web apps
- Apply extremely well to docker apps
- Rule #3: **Store config in the environment**
- The blog app is a 12-factor app
- We have to pass:
  - A SECRET environment variable
  - a MONGODB_URI environment variable

For more information: https://12factor.net

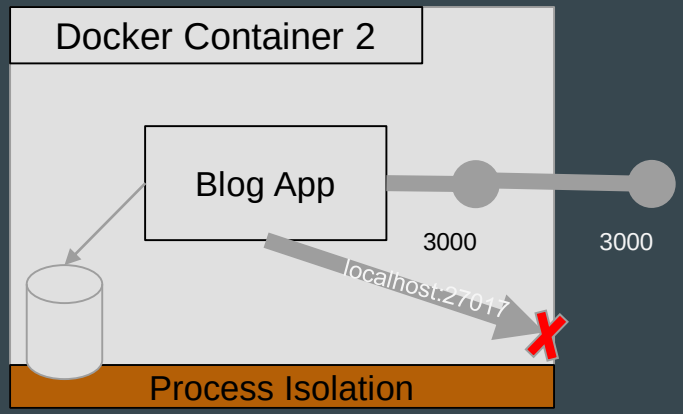
Running the App

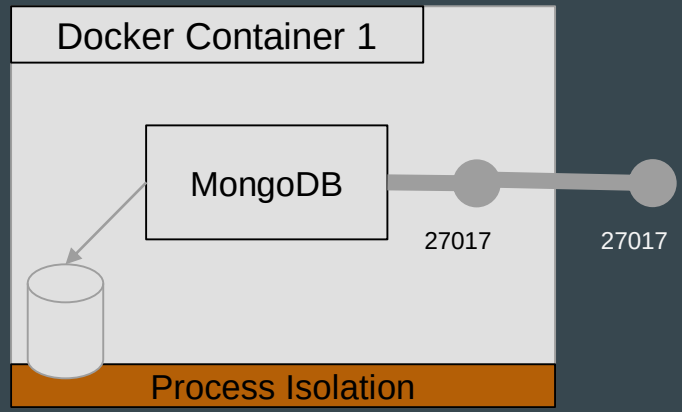
```
$ docker run -t -p 3000:3000 \
        -e SECRET=shhhh \
        -e MONGODB_URI=localhost:27017 \
        giltayar/blog-app-example


...
Error: connect ECONNREFUSED 127.0.0.1:27017
    at Object._errnoException (util.js:1003:13)
```

So I can Ctrl+C

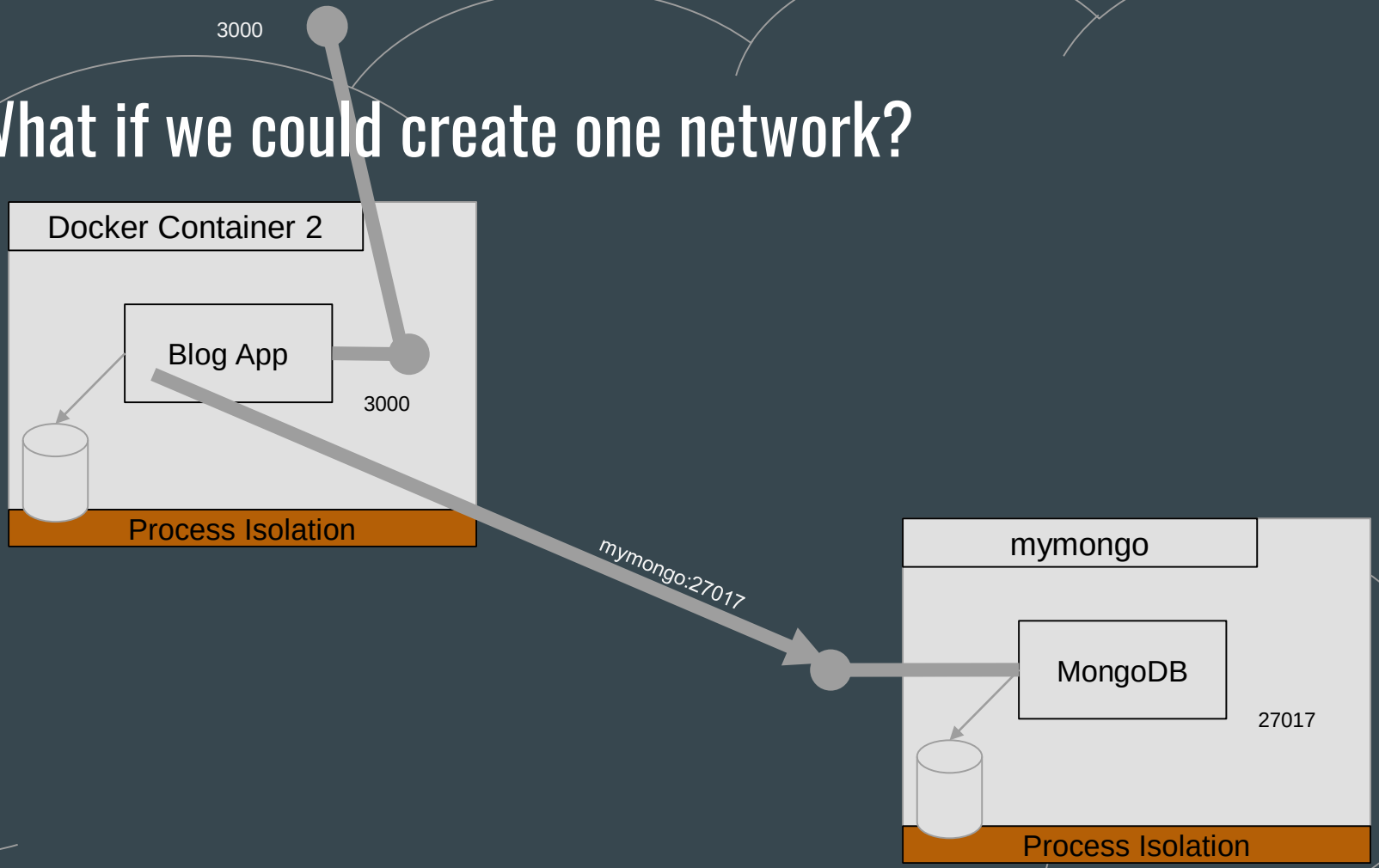
Environment variable

localhost in a container refers to the container itself



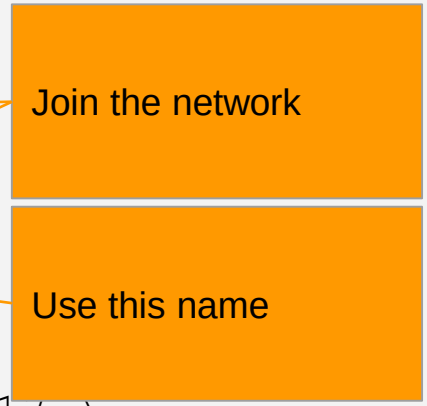


What if we could create one network?



Running the App in a network

```
$ docker network create mynet

$ docker run -d \
      --network=mynet \
      --name=mongo \
      mymongo

$ docker run -t -p 3000:3000 \
      -e SECRET=shhhh -e MONGODB_URI=mongo:27017 \
      --network=mynet \
      myblog
```



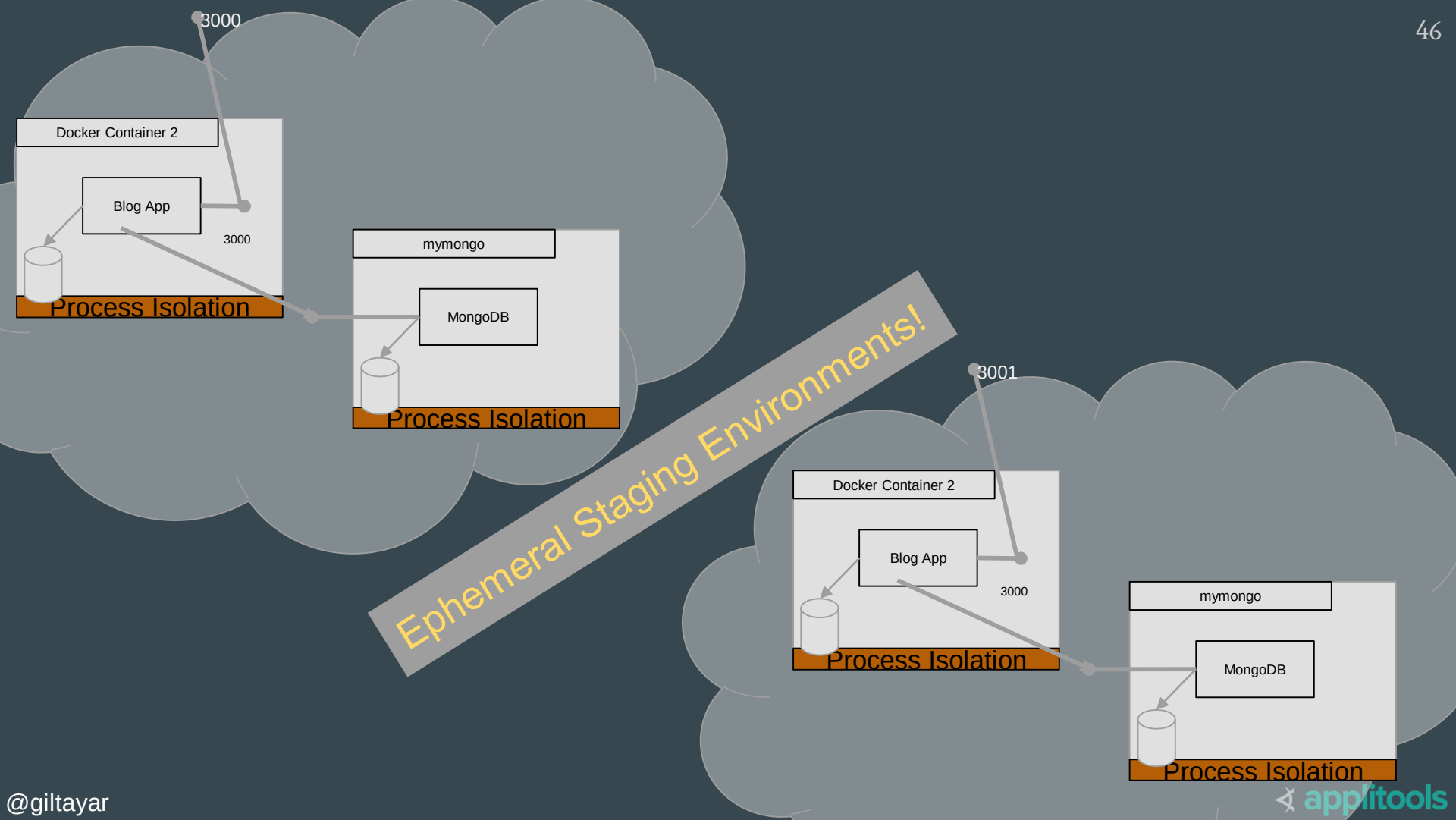
Running the same app *twice*

```
$ docker network create mynet

$ docker run -d --network=mynet --name=mongo mymongo
$ docker run -d -p 3000:3000 --network=mynet \
    -e SECRET=shhhh -e MONGODB_URI=mongo:27017 myblog

$ docker network create mynet2

$ docker run -d --network=mynet2 --name=mongo2 mymongo
$ docker run -d -p 3001:3000 --network=mynet2 \
    -e SECRET=shhhh -e MONGODB_URI=mongo2:27017 myblog
```

Let's try them...

http://localhost:3000

http://localhost:3001


3000
Docker Container 2
Blog App
3000
Process Isolation
mymongo
MongoDB
Process Isolation
Ephemeral Staging Environments!
3001
Docker Container 2
Blog App
3000
Process Isolation
mymongo
MongoDB
Process Isolation

What if there was an even simpler way to run multiple containers?

Docker Compose!

Docker Compose

- Write a file (docker-compose.yml)
- The file declares all the containers and the connections between them
- Then just “run” the docker composition

Let’s do it!

docker-compose.yml

```yaml
version: '3'
services:
  mongo:
    image: mymongo

  blog:
    image: myblog
    environment:
      SECRET: shhh
      MONGODB_URI: mongo:27017
    ports:
      - "3000:3000"
```

Running it

```
$ docker-compose up
```

or…

```
$ docker-compose up -d
```

Trying it

http://localhost:3000

Killing it

```
$ docker-compose stop
```

or…

```
$ docker-compose down
```

Can we run two instances of a docker-compose?

- Yes, but we need to take care not to use the same host port.
- Time, alas, does not permit me to show you the details
- You can check the git repo for the full information

OK, this is cool.
But what has this got to do with Staging?

docker-compose

- Enables you to run the whole application locally
- Does *not* run your application in a staging environment...
- ... or in production

So whadda we do?


LEVEL UP!

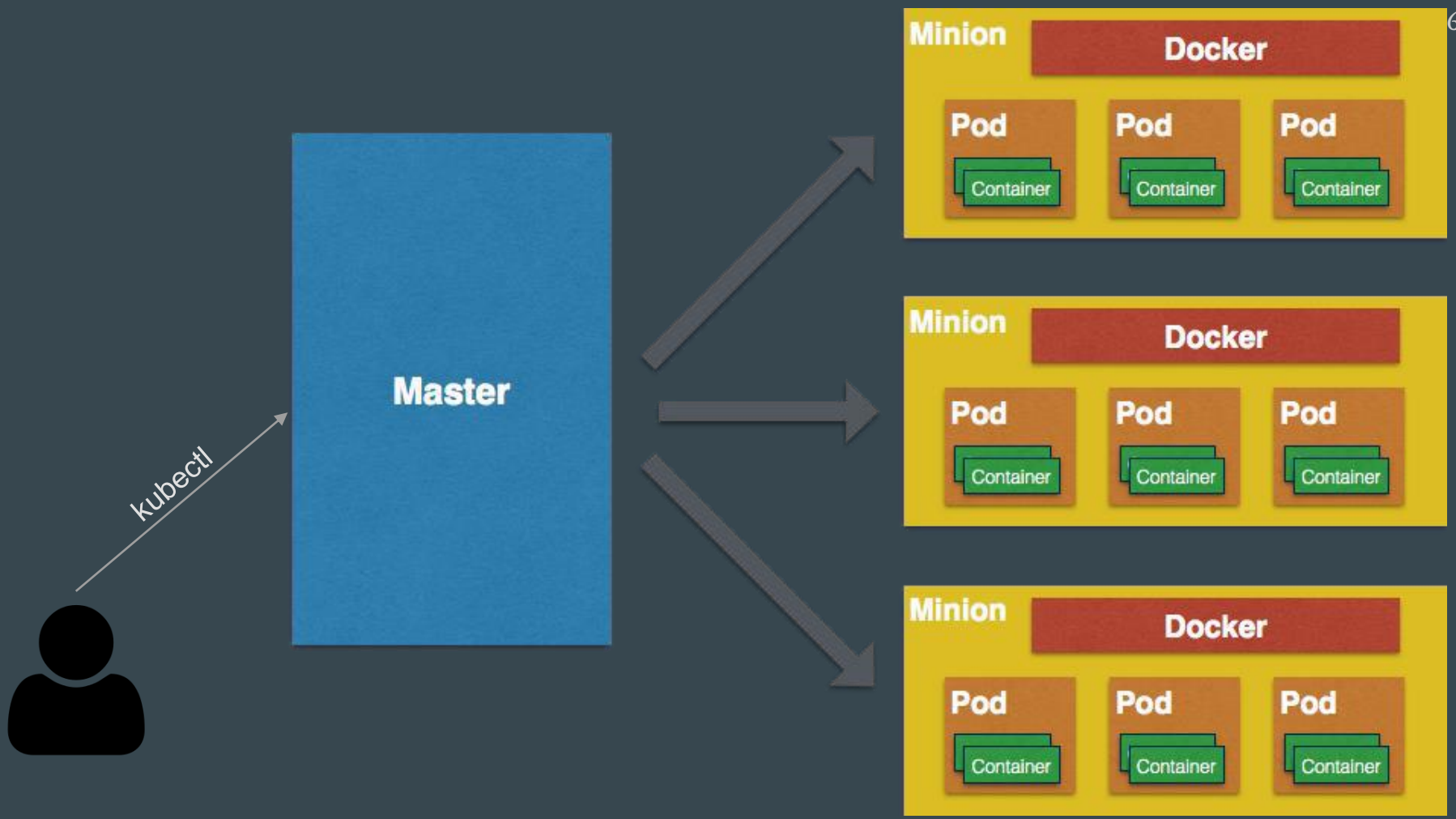
kubernetes

Kubernetes is docker-compose for production

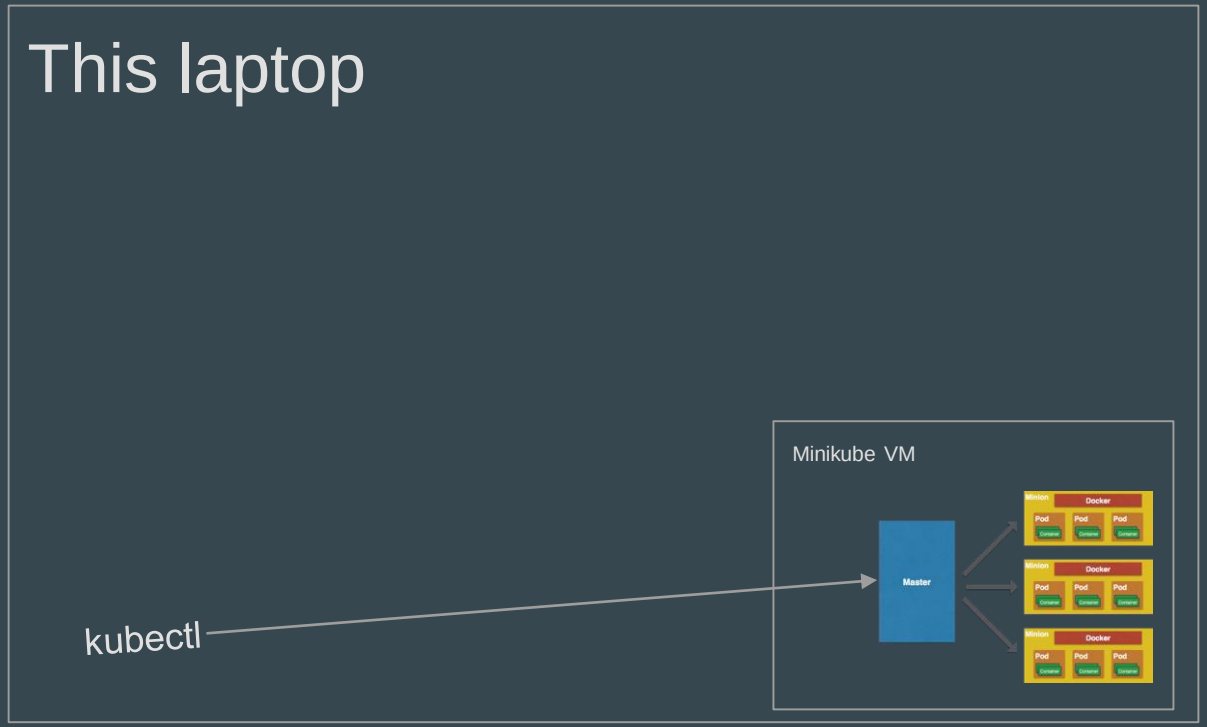
Kubernetes

- Like Docker-compose:
  - Also uses docker containers to run services
  - Can also deploy easily using a set of declarative files
  - Easy to use

- But better:
  - Can deploy multiple services to multiple computers
  - Self-heals itself
  - Robust enough to be used in production


Master
kubectl
Minion
Docker
Pod
Container
Pod
Container
Pod
Container
Minion
Docker
Pod
Container
Pod
Container
Pod
Container
Minion
Docker
Pod
Container
Pod
Container
Pod
Container

Minikube (or... it might as well have been in the cloud)



myblog-1.yml

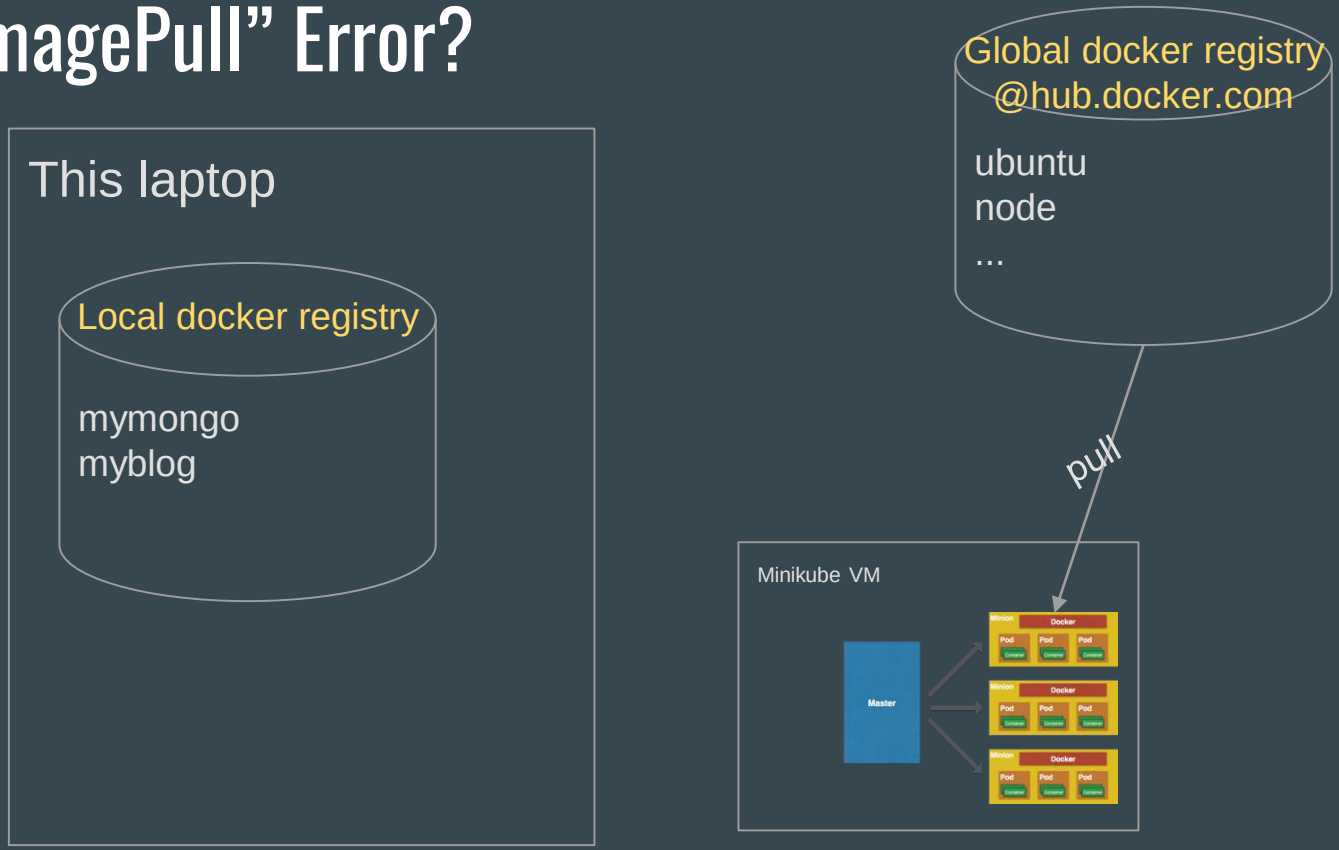
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: myblog
spec:
 selector:
   matchLabels:
     app: myblog
 replicas: 1
 template:
   metadata:
     labels:
       app: myblog
```

```yaml
  spec:
    containers:
    - name: myblog
      image: myblog
      ports:
      - containerPort: 3000
      env:
        - name: MONGODB_URI
          value: 'mymongo:27017'
        - name: SECRET
          value: 'shhhhhh'
```

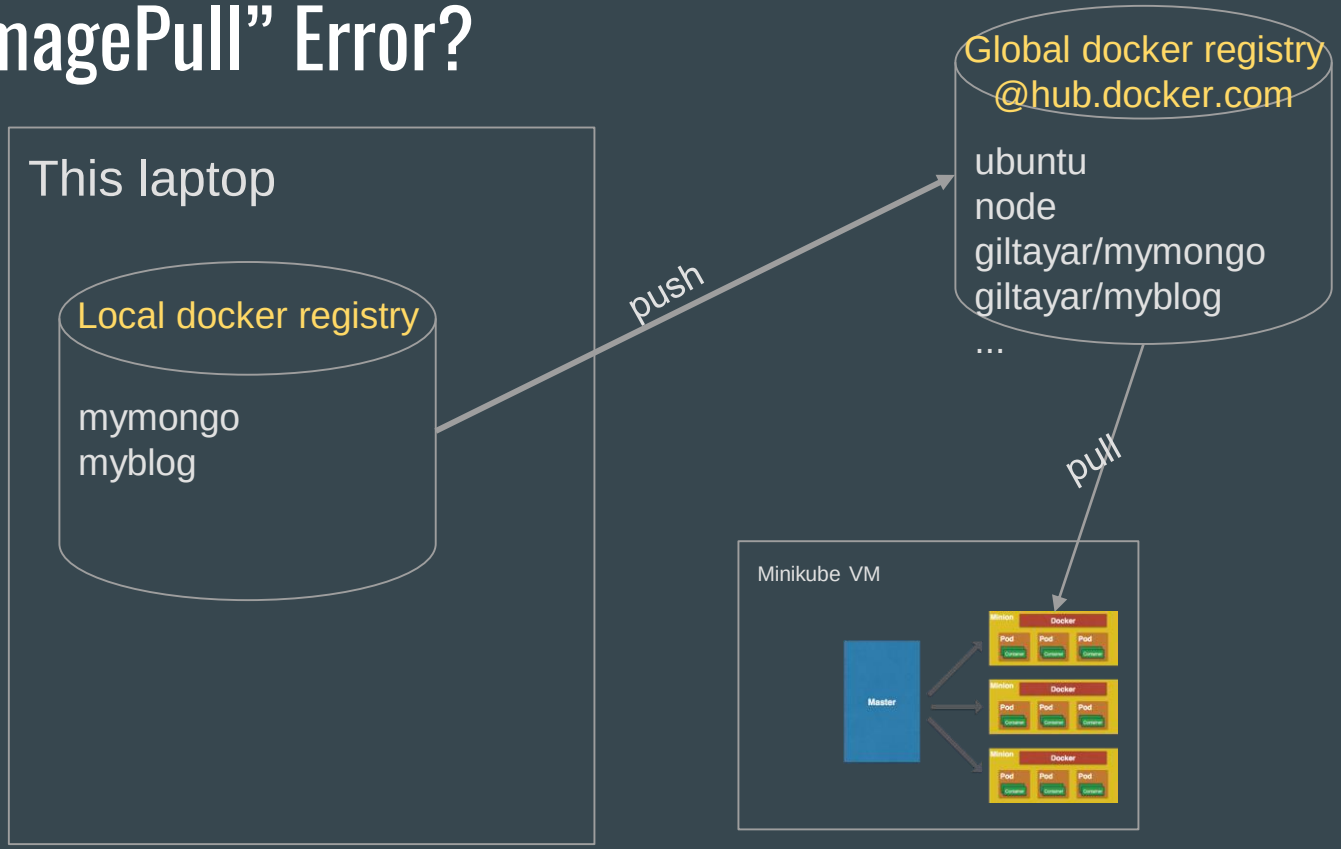
Let's "apply" the yml to the Kubernetes cluster

```
$ kubectl apply -f myblog-1.yml
deployment "myblog" created
$ kubectl get pods
NAME                      READY     STATUS          RESTARTS   AGE
myblog-56589c8f7f-xdlgz   0/1       ErrImagePull    0          5s
```

Why the “ErrImagePull” Error?



Why the “ErrImagePull” Error?



So let's push our images to the global docker registry

```
$ docker tag myblog giltayar/myblog
$ docker push giltayar/myblog


$ docker tag mymongo giltayar/mymongo
$ docker push giltayar/mymongo
```

myblog-2.yml (fixed)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: myblog
spec:
 selector:
   matchLabels:
     app: myblog
 replicas: 1
 template:
   metadata:
     labels:
       app: myblog
```

```yaml
spec:
  containers:
  - name: myblog
    image: giltayar/myblog
    ports:
    - containerPort: 3000
    env:
      - name: MONGODB_URI
        value: 'mymongo:27017'
      - name: SECRET
        value: 'shhhhhh'
```

Let's reapply...

```
$ kubectl apply -f myblog-2.yml

$ kc get pods
NAME                        READY     STATUS             RESTARTS   AGE
myblog-6b4567d975-h89g6     0/1       CrashLoopBackOff   3          3m

$ kc logs myblog-6b4567d975-h89g6
...
Error: getaddrinfo ENOTFOUND mymongo mymongo:27017
```

mymongo.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: mymongo
spec:
 selector:
   matchLabels:
     app: mymongo
 replicas: 1
 template:
   metadata:
     labels:
       app: mymongo
```

```yaml
spec:
  containers:
  - name: mymongo
    image: giltayar/mymongo
    ports:
    - containerPort: 27017
```

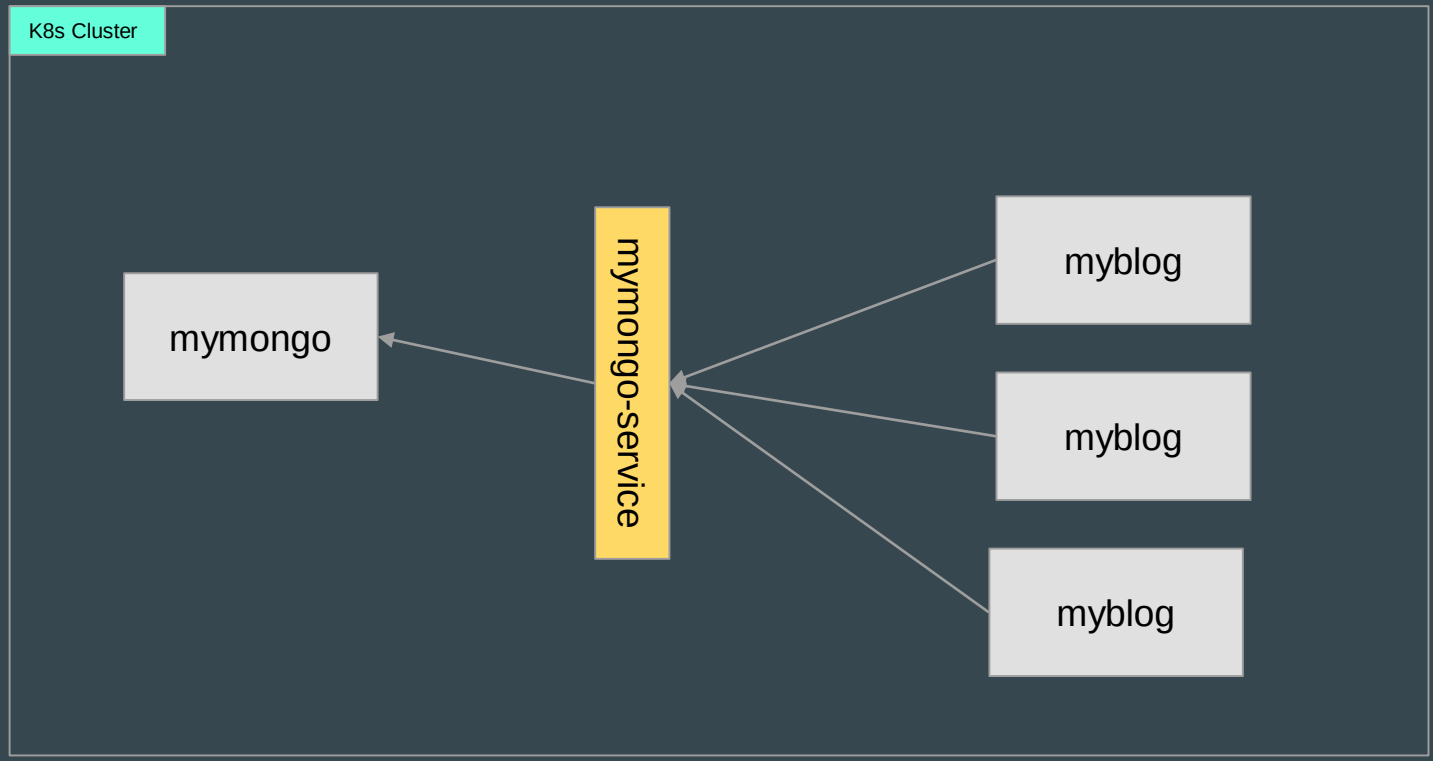
Let's apply the mongo yml

```
$ kubectl apply -f mymongo.yml

$ kc get pods
NAME                        READY     STATUS    RESTARTS   AGE
myblog-6b4567d975-h89g6     0/1       Error     6          7m
mymongo-67b9bd8c7f-87j2j    1/1       Running   0          2m


$ kc logs myblog-6b4567d975-h89g6
...
Error: getaddrinfo ENOTFOUND mymongo mymongo:27017
```

Same Problem. Why?

- Because in docker-compose,
  the name of the service is also the way to discover it
- In kubernetes, a deployment is just a set of containers.
  - Not discoverable
- To “discover” pods, create a **Kubernetes Service**

Pods are not discoverable



We need a service

mymongo-service.yml

```
kind: Service
apiVersion: v1
metadata:
 name: mymongo-service
spec:
 selector:
   app: mymongo
 ports:
 - protocol: TCP
   targetPort: 27017
   port: 27017
```

myblog.yml (fixed again)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: myblog
spec:
 selector:
   matchLabels:
     app: myblog
 replicas: 1
 template:
   metadata:
     labels:
       app: myblog
```

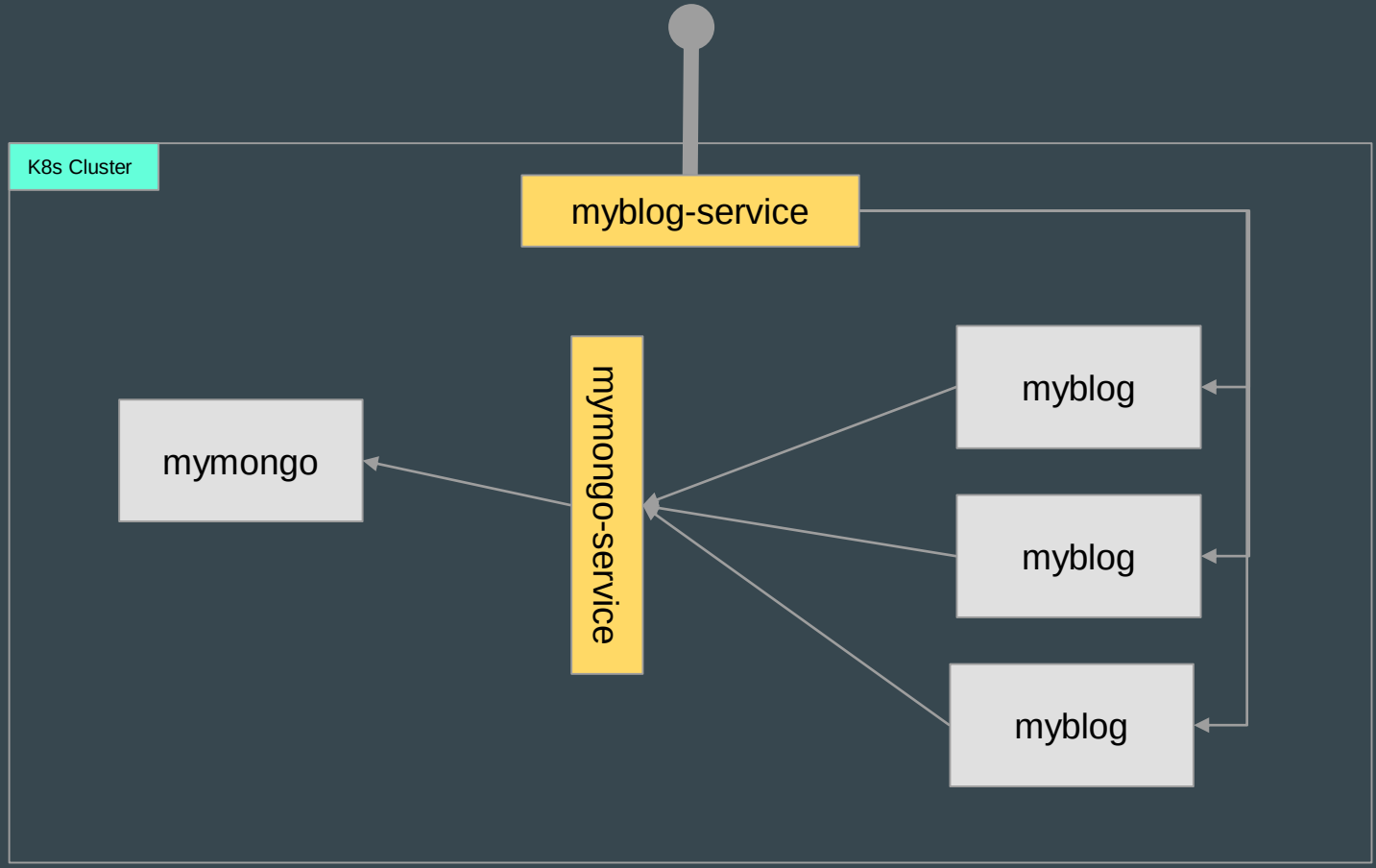
```yaml
spec:
  containers:
  - name: myblog
    image: giltayar/myblog
    ports:
    - containerPort: 3000
    env:
      - name: MONGODB_URI
        value:
          'mymongo-service:27017'
      - name: SECRET
        value: 'shhhhhh'
```

And now applying...

```
$ kubectl apply -f mymongo-service.yml
service "mymongo-service" created

$ kubectl apply -f myblog.yml
deployment "myblog" configured

$ kc get pods
NAME                       READY     STATUS    RESTARTS   AGE
myblog-566ccc499c-sx8cf    1/1       Running   0          2m
mymongo-67b9bd8c7f-87j2j   1/1       Running   0          14m
```

Blog is running. Can we access it?

- No! It's just a set of containers.
- We need a service.
- But this time, the service needs to be accessible from outside the cluster!



We need a service

myblog-service.yml

```
kind: Service
apiVersion: v1
metadata:
 name: myblog-service
spec:
 selector:
   app: myblog
 ports:
 - protocol: TCP
   targetPort: 3000
   port: 3000
 type: NodePort # this is how you enable access from outside
```

And now applying...

```
$ kubectl apply -f myblog.yml
service "myblog-service" created

$ kubectl apply -f myblog.yml
deployment "myblog" configured

$ kubectl get services
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
myblog-service    NodePort    10.109.42.144    <none>        3000:31489/TCP
mymongo-service   ClusterIP   10.103.150.100   <none>        27017/TCP
```

And now we try it...

```
$ minikube ip

192.168.64.8
```

http://192.168.64.8:31489/

And it works!

- And the same deployment procedure will work in production...

Let's create another parallel environment

```
$ kubectl create namespace blog1


$ kubectl -n blog1 apply -f \
mymongo.yml,\
Mymongo-service.yml,\
myblog.yml,\
myblog-service.yml
```

Summary

- The long journey...
- Running individual containers using docker
- Running environments locally using docker-compose
- Running environments in a production environment using Kubernetes

This is how you build your staging environment:

- Docker images built by developers...
- ... and deployed into Kubernetes clusters

Thank You!

...

@giltayar
This presentation: http://bit.ly/docker-and-the-path-starcanada
Github repo: https://github.com/giltayar/docker-and-the-path