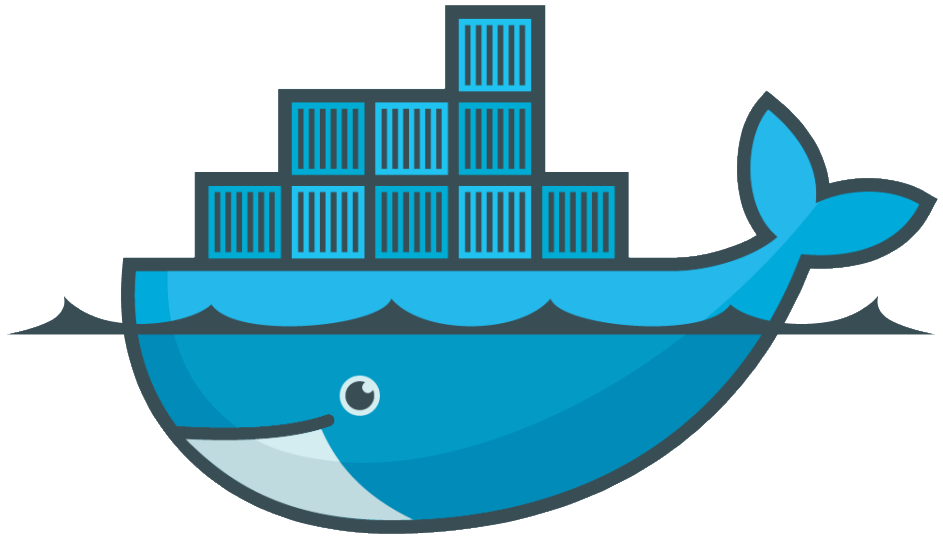
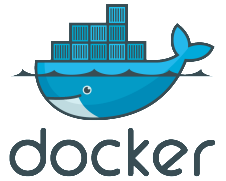


BY DAVID ELNER

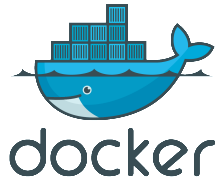
APRIL 2017

Some important questions…



1. What is Docker? What isn't it?
2. What advantages does it offer?
3. How does it work?
4. How it can it help us?
5. How can we implement it?

But first…

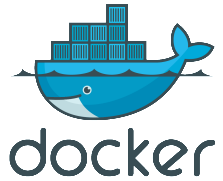


This presentation includes some exercises to follow along with. Before we continue, you should…

1. Install Docker on your machine
   https://docs.docker.com/engine/installation/

2. Clone the Docker Training repo
   https://github.com/delner/docker-training

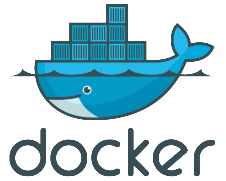
But first why Docker?



Some common problems developers have…

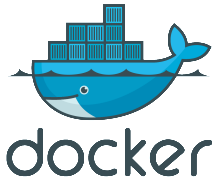
- Setting up new applications locally is difficult
- Sometimes you installed the wrong version of the right software (e.g. Postgres, etc.)
- Configuring deployment for new software is difficult
- Inconsistent development environments can make it difficult to debug issues

Docker can help solve these



- Application setup can be done in a single command
- Application requirements can be ‘versioned’ and shared
- Configuring deployment of a new service doesn’t require expert knowledge of that service
- An entire suite of microservices can be started locally with a single command

Some cool things about Docker…

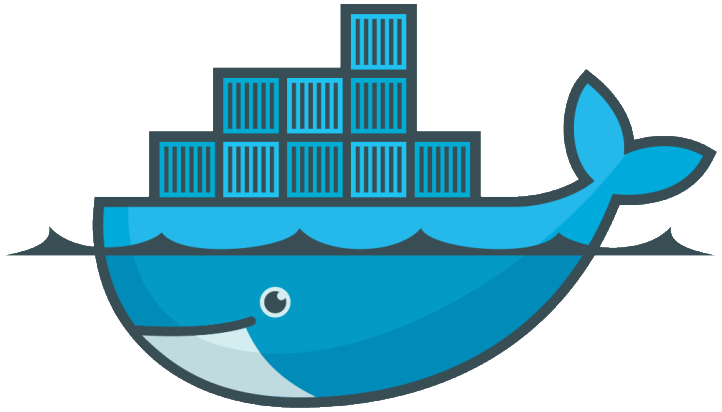


1. **Applications become tiny packages**
    1. Version controlled and shareable
    2. Anyone can download and run instantly

2. **Application environments are identical across hosts**
    1. Packages contain their own libs/binaries
    2. Consistent behavior (e.g. Yosemite vs El Capitain, OSX vs Linux)

3. **Run multiple applications side-by-side on single host**
    1. Isolated applications allows concurrent execution
    2. Start a suite of services with one command `**docker-compose up**`

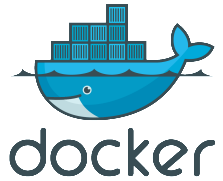
What is it?



docker

A VIRTUAL MACHINE? A SERVICE? A WHALE?

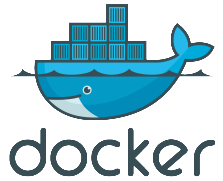
What it is…



Docker is a daemon that hosts other applications.

It allows developers to run applications, in isolation, natively on a single host machine.

What it is…



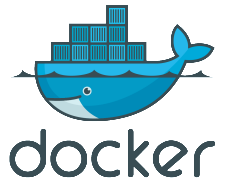
For new users familiar with VMs, think of Docker **like***

a tool for running virtual machines.

* You'll use it this way, but it technically functions a bit differently.

However…

It is not...

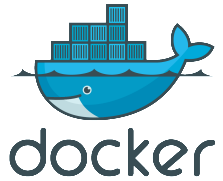


- **…a virtual machine, or virtualization framework.**
  - Like Vagrant, VirtualBox, VMWare
  - (Runs in a VM on OSX and Windows hosts.)

- **…SaaS, or other business service.**
  - Like Elastic Container Service (ECS), Docker Cloud, etc.

- **…a deployment or orchestration framework itself.**
  - Like Marathon, Consul, Puppet, or Chef
  - Docker Swarm provides something similar to this

Native vs Virtualization

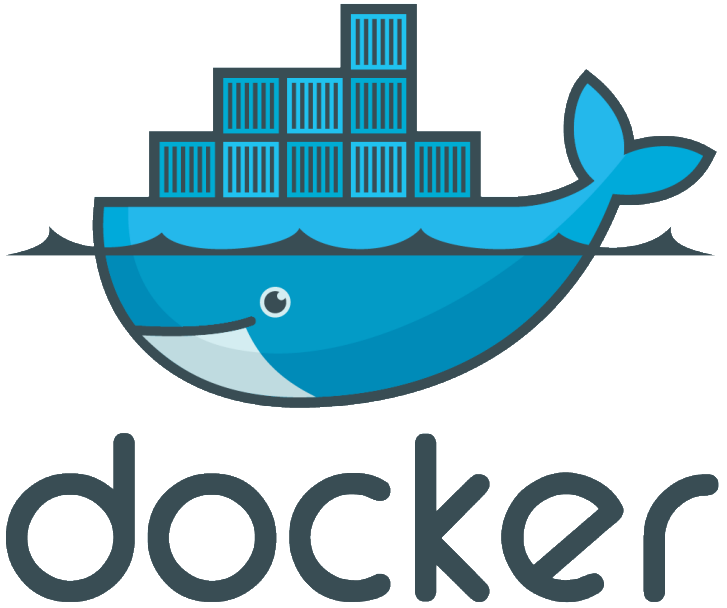


- **For most practical applications, you use Docker like a virtual machine.**
  - Much of your workflow will be similar to a VMWare/VirtualBox setup.

- **Under the hood, Docker does not use virtualization, but native processes.**
  - It translates program instructions to host kernel instructions.
  - Think of the Docker Engine like the Java Runtime Environment (JRE).
  - It makes it run super-fast. (No virtualized hardware.)
  - It allows host to optimally use resources (CPU/memory) vs VM solutions.

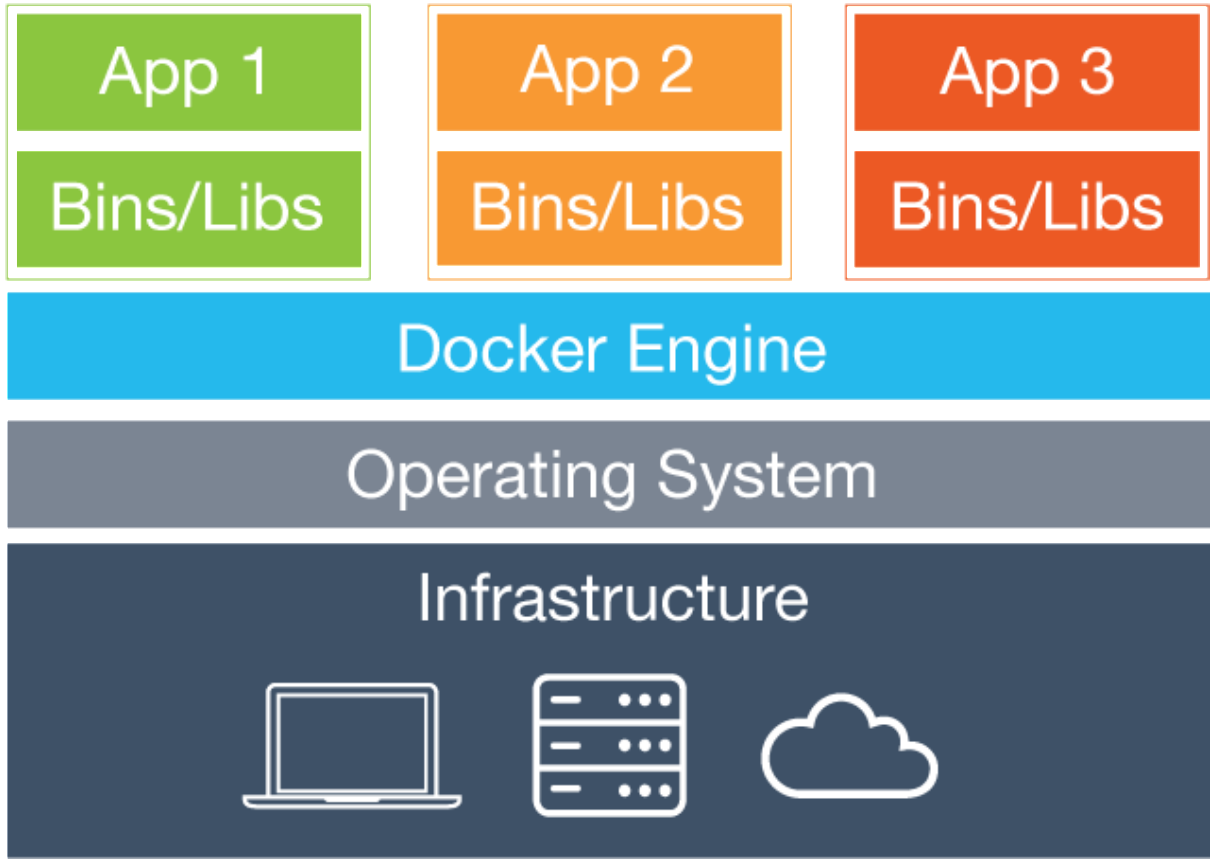
"Docker is to Virtualization, as a Compiled language is to a Scripted language."

How does it work?

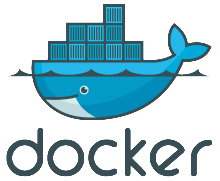


LET'S TALK ABOUT THE BASICS

What it is… the Docker Engine



Some basic concepts



Images

Snapshot of a machine's filesystem.
(Equivalent to image for a VM.)

Containers

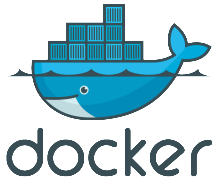
A running copy of an image.
(Equivalent to VM.)

"Images are to containers, as classes are to objects."

Images

WHAT THEY ARE AND HOW THEY WORK

Images

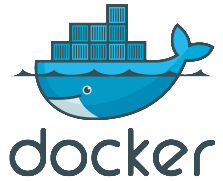


- **Images** are “snapshots” of a file system
  - They are binary files.
  - They contain files, binaries and libraries added at “build time.”
  - Images that are just an operating system are **base images**

Base Images

fedora ***centos*** ***ubuntu*** ***debian*** ***busybox***

Image tags



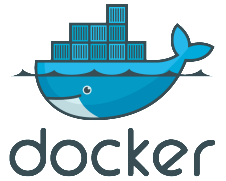
- **Images** are versioned.
  - When built, they are given a name and tag.
  - When working with them, reference by their name and tag.
  - If you don't specify a tag, it defaults to ***latest***.

<name>:<tag>

centos → ***centos:latest***

ubuntu:16.04

DockerHub



- You can find images to use through **DockerHub**
  - **DockerHub** is to **Docker** as **GitHub** is to **Git**
  - Has thousands of images in public repositories you can use
  - Docker CLI automatically searches DockerHub for you
  - Reference a specific image on DockerHub by:

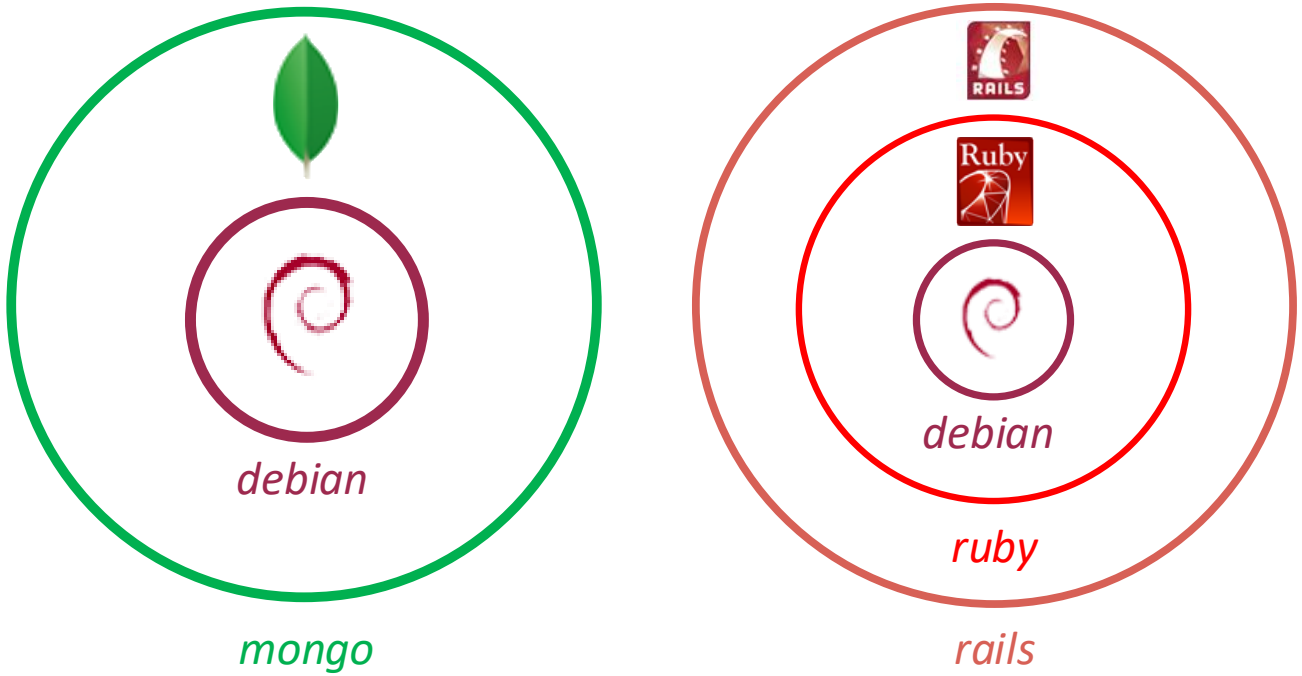
<user or org>/<name>:<tag> → ***delner/ubuntu:16.04***

More sophisticated images

"Images are like onions." – Shrek.

More sophisticated images

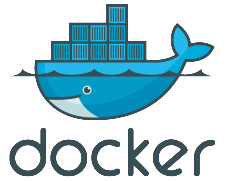
- Images are layered upon base images, or other images.
- Take these images from DockerHub...



Containers

WHAT THEY ARE AND HOW THEY WORK

Containers

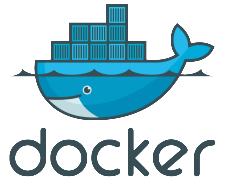


- **Containers** are running copies of an image
  - They can be started, stopped, or killed.
  - They are practically used just like virtual machines.

Think mini-server!

```
~/src/docker/expressApp$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
b792aa955ec7        node:latest         "npm start"              6 seconds ago       Up 5 seconds
58fa1571f67a        node:0.10           "node --version"         23 seconds ago      Exited (0) 21 seconds ago
8c50d1e7d7e9        node:latest         "node --version"         30 seconds ago      Exited (0) 28 seconds ago
```

Containers

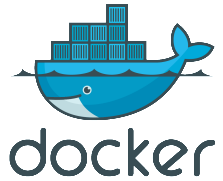


Although containers are **like*** virtual machines,

they are NOT virtual machines.

*Similar in purpose, dissimilar in application.

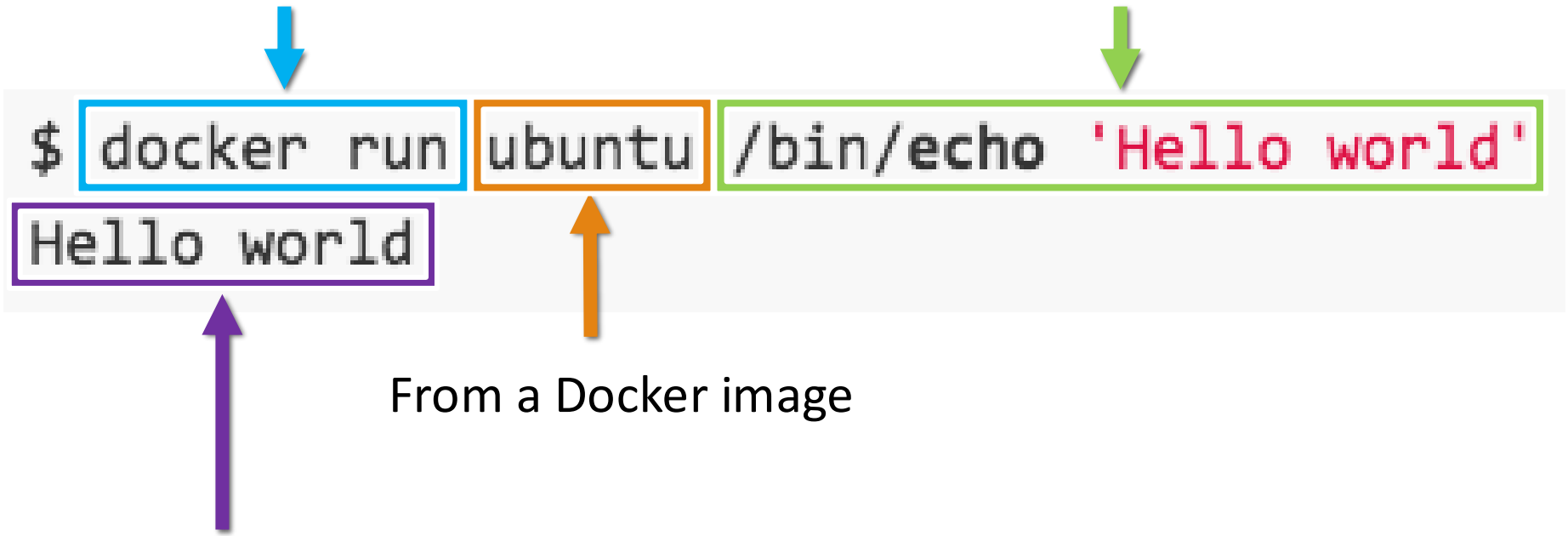
Containers vs VMs



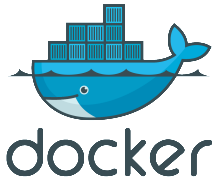
- **Unlike a virtual machine, Docker does NOT run full-blown VMs.**
  - Docker ”VMs”, aka **containers**, only run a single process from an image
    - E.g. A container might run **nginx –g daemon off** to run a web server from the *nginx* image
  - **Containers** do NOT run indefinitely: they live only as long as that process runs
    - When the command completes, the container stops.

```
~/src/docker/expressApp$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
b792aa955ec7        node:latest         "npm start"         6 seconds ago       Up 5 seconds
58fa1571f67a        node:0.10           "node --version"    23 seconds ago      Exited (0) 21 seconds ago
8c50d1e7d7e9        node:latest         "node --version"    30 seconds ago      Exited (0) 28 seconds ago
```

Running a Docker container



Containers



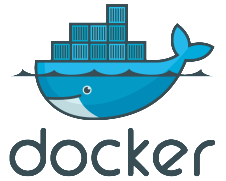
EXERCISE #1:

Pulling images
& running containers

Custom images

HOW TO MAKE THEM

Custom images

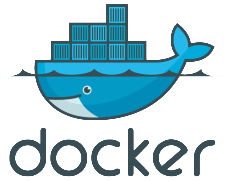


Pulling and running default images is cool and all…

…but not really that useful in itself.

To run our own software, we need to build our own images on top of these.

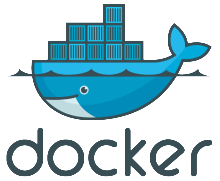
Custom images



Two ways to make our own:

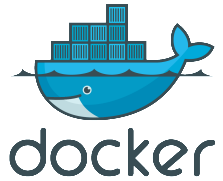
1. Modify an existing image, save the changes
2. Create a new image

Changing an existing image

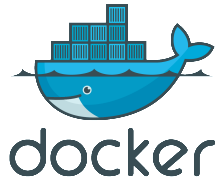


1. **Pull an image** you want to modify.
2. **Create a container** from the image.
3. **Change the filesystem** of that container.
4. **Commit state** of container as a new image.
5. (Optional) **Tag new image** with name & version.

Changing an existing image



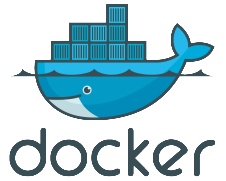
1. `**docker pull** ruby:2.3.3`
2. `**docker run** ruby:2.3.3 gem install rails`
3. `**docker commit** –m ‘Rails’
-a ‘David’
<container ID>
delner/rails:5.0.2



EXERCISE #2:

Changing an existing image.

Creating a new image...



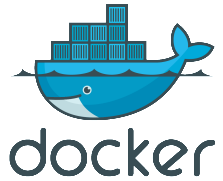
You can create a **Dockerfile**, which contains specific instructions on how to build an image.

- Each **RUN** statement is like running a command in BASH within the container
- The filesystem changes resulting from each **RUN** statement becomes a commit
- All commits get baked into the new image when built.

Think layer in an onion!

```
1  # Dockerfile
2  FROM training/sinatra
3  MAINTAINER David Elner <david@davidelner.com>
4
5  RUN gem install json
```

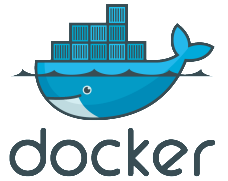
docker build -t delner/sinatra:v2 .



EXERCISE #3:

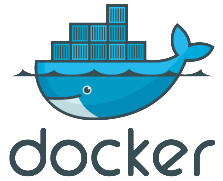
Building a new image.

Sharing images on DockerHub



- Like **GitHub**, great for sharing images privately and publicly
  - Sign up for DockerHub account
  - Create your own repositories for your own images
  - Push/commit/pull through Docker CLI, just like Git

Sharing images on DockerHub



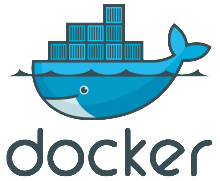

```
$ export DOCKER_ID_USER="username"
```

```
$ docker login
```

```
$ docker tag my_image $DOCKER_ID_USER/my_image
```

```
$ docker push $DOCKER_ID_USER/my_image
```

Containers



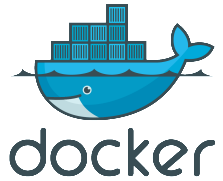
EXERCISE #4:

Sharing images.

Volumes

HOW TO PERSIST DATA

Changes are ephemeral!

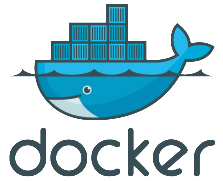


- Any changes you make to acontainer are temporary.
  - After stopping a container, you can **docker commit** to save them.
  - Otherwise killing a container will discard them.

- What about databases? How do I persist data between sessions?
  - Baking application data into your image with **docker commit** is frowned upon.
  - Instead, use **volumes**. They're directories you can mount onto containers.

Think USB thumb drive.

Managing volumes

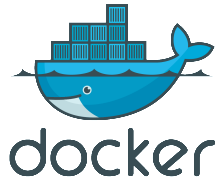


$ docker volume ls

$ docker volume create <Vol name>

$ docker volume rm <Vol name/Vol ID>

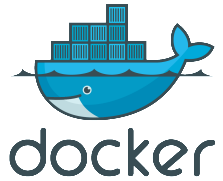
Mounting volumes/directories



$ docker run –v myvolume:/home/root/ ubuntu:16.04

$ docker run –v ~/docker-training:/home/root/ ubuntu:16.04

Volumes



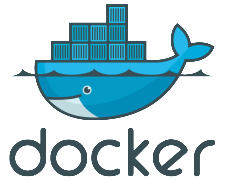
EXERCISE #5:

Volumes.

Networking

HOW TO CONNECT CONTAINERS

Networking between containers

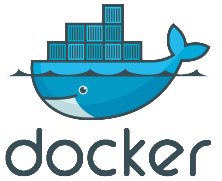


Every container is *isolated.* They act like their own virtual machines with their own filesystems.

We need to network them if we want them to communicate…

Docker provides networking between containers.

Listing networks

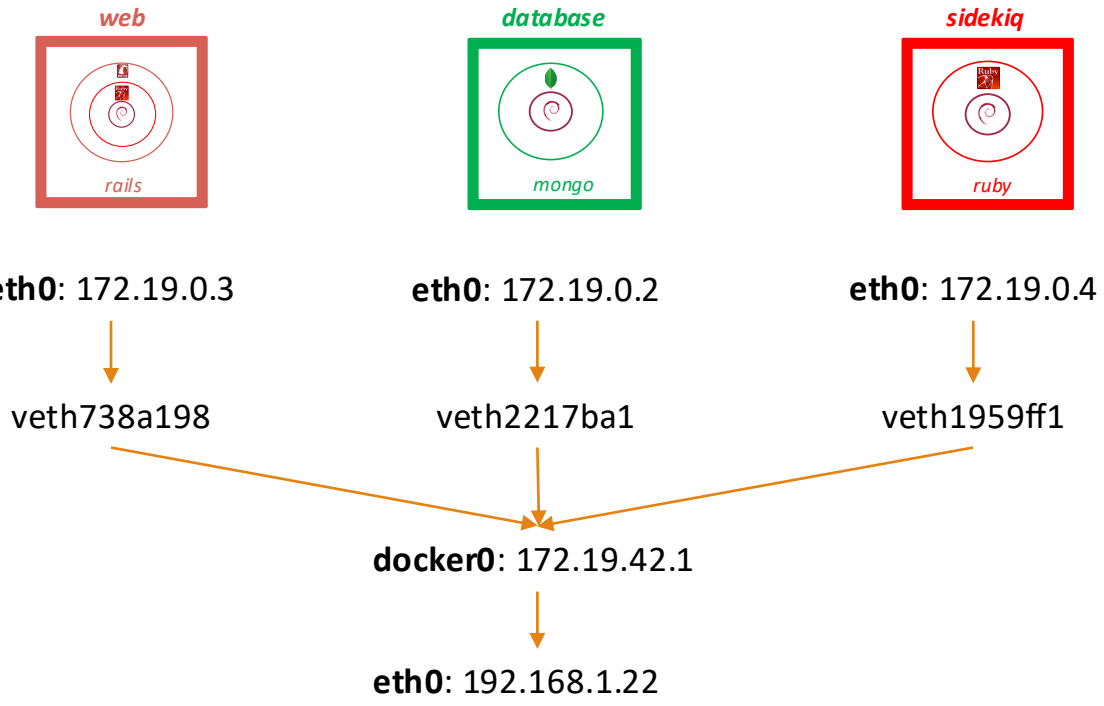


$ docker network ls

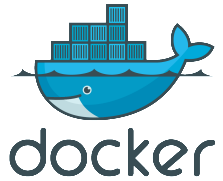
```
david:docker-training david.elner$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
772b52febb9e        bridge              bridge              local
c2a8f4f0a5cf        host                host                local
5e27b79580af        none                null                local
```

- Docker has some default networks: ***bridge***, ***host***, ***none***.
- Docker adds containers to the ***bridge*** network by default.
  - Containers in this network CANNOT find each other by default.

Default bridge network



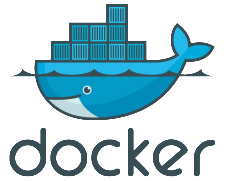
Creating networks



- You can create a custom network to add containers to.
  - Containers in these networks are discoverable by name to one another.
  - But NOT to containers outside the network.
  - Great for networking applications built from multiple containers.

$ docker network create –attachable myapp

Adding containers to a network



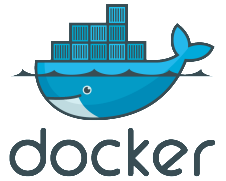
When starting containers, pass the *–network* and *–name* flag.

$ docker run --network myapp --name web -d rails

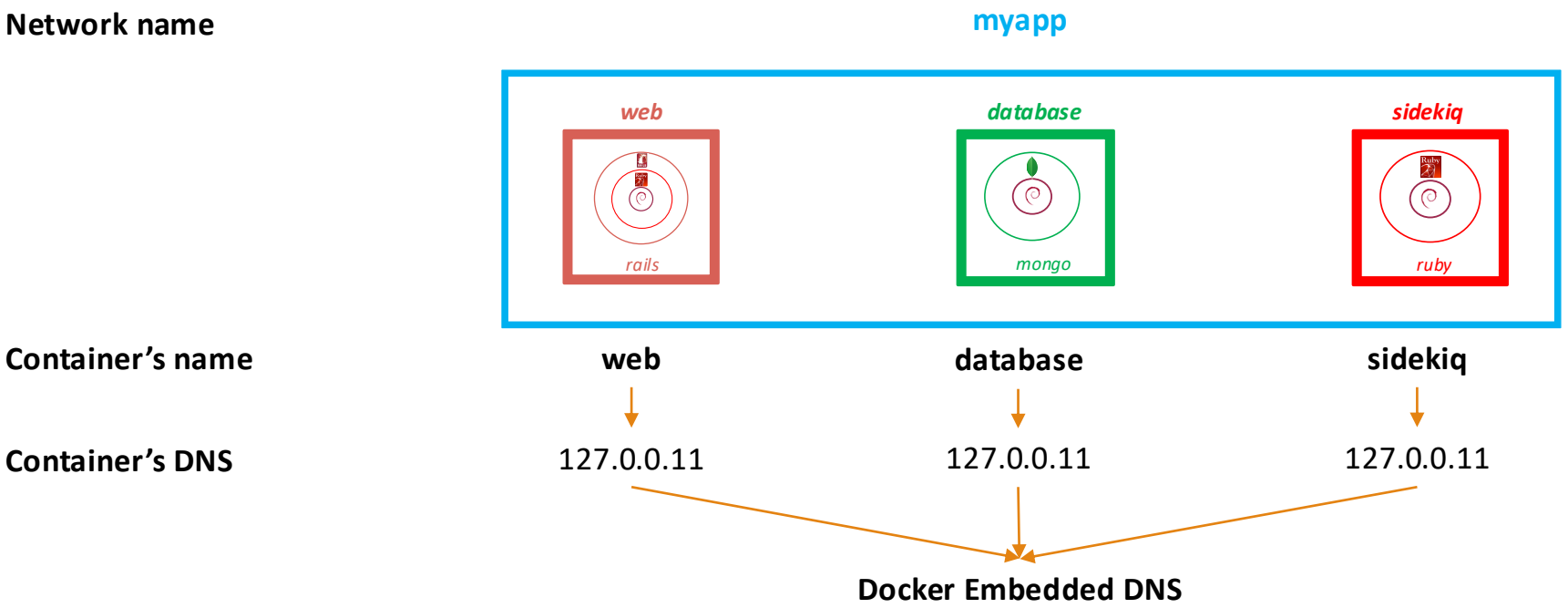
$ docker run --network myapp --name database -d postgres

$ docker run --network myapp --name sidekiq -d ruby

Networking between containers

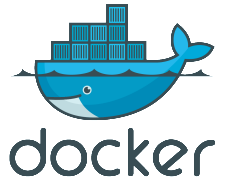


Use Docker's *embedded DNS* to refer to other containers by name.




```
root@5718c85094b1:/app# ping database
PING database (172.19.0.2): 56 data bytes
64 bytes from 172.19.0.2: icmp_seq=0 ttl=64 time=0.113 ms
```

Exposing ports



Despite being discoverable, container's ports will be inaccessible. Open ports by passing the ***–p*** flag to the *docker run* command.

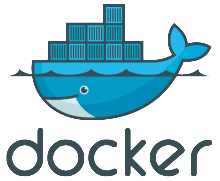
$ docker run --network myapp –name web –p 3000 -d rails

$ docker run --network myapp --name database –p 5432 -d postgres

Use *<host port>:<container port>* format to bind to host port.

$ docker run --network myapp –name web –p 80:3000 -d rails
$ curl localhost:80

Networking



EXERCISE #6:

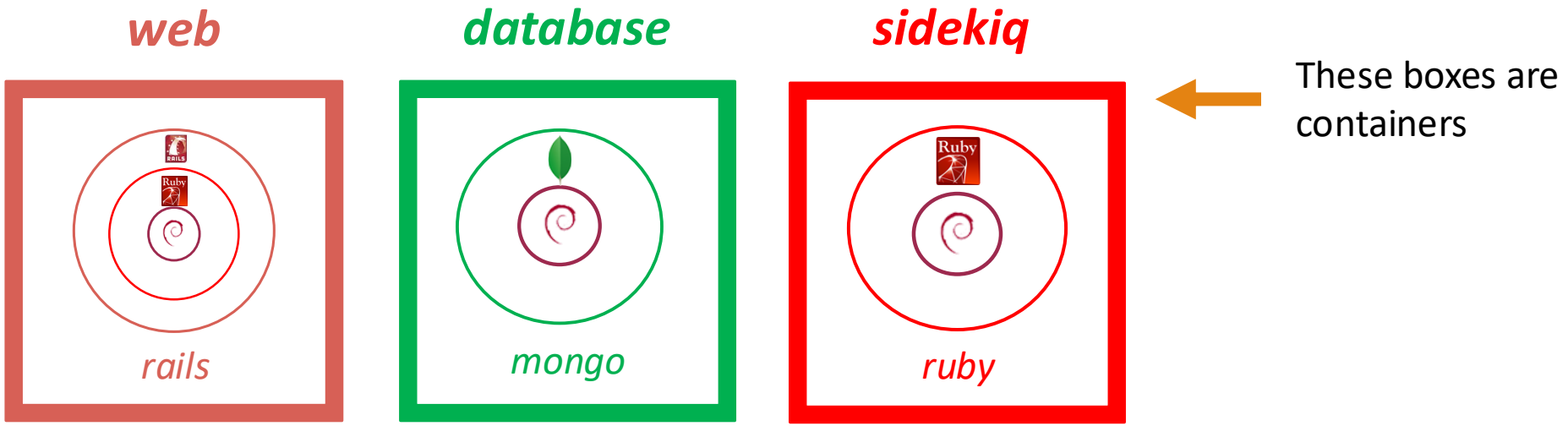
Networking.

Part 2 (to be continued)

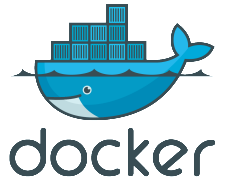
DOCKER COMPOSE AND DOCKERIZATION

Images and Containers

Let’s look at sample suite of microservices…



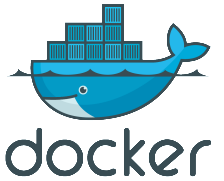
Running multiple containers...



Let’s say we want to run a web container and database…

docker-compose up starts both containers

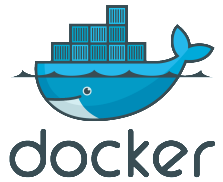
Running multiple containers...



You can create a **docker-compose.yml**, which defines some well known containers.

```
1  # docker-compose.yml
2  web:
3    container_name: sinatra-web
4    build: . # Builds image from 'Dockerfile'
5  db:
6    container_name: mongo-db
7    image: mongo # Pulls 'mongo' image
```

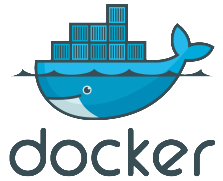
docker-compose up starts both containers



But I want to run my app in Docker?

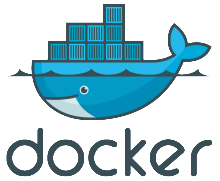
HOW TO “DOCKERIZE” AN APPLICATION

Let’s explore by example



- Let’s say we have a Rails application
  - For simplicity, it doesn’t use a database.

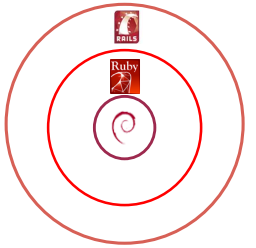
Creating an image



- Add a **Dockerfile** to the project
  - Creates an image that bakes in our gems, but not our application files.

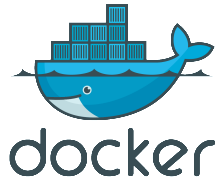
```
1  # docker-demo/Dockerfile
2  FROM rails:4.2.6
3  MAINTAINER David Elner <david@davidelner.com>
4
5  # Setup a working directory
6  WORKDIR /app
7
8  # Bundle any gems we need
9  COPY Gemfile /app
10 COPY Gemfile.lock /app
11 RUN bundle install
```

Think onion!



rails

Configuring a container

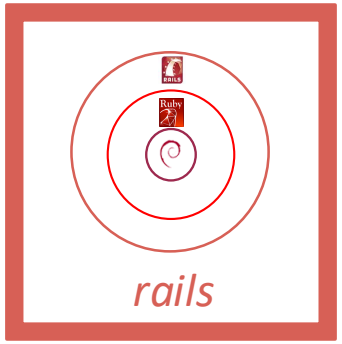


- Add a **docker-compose.yml** file to the project
  - Defines what containers 'compose' your application

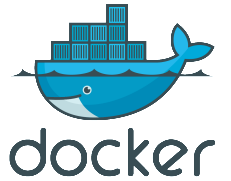
```
1  version: '2'
2  services:
3    web:
4      container_name: docker-demo-web
5      build: .
6      volumes:
7        - .:/app
8      command: rails server -b 0.0.0.0
9      ports:
10       - "3000:3000"
```

Think box!

web



Build and run the image

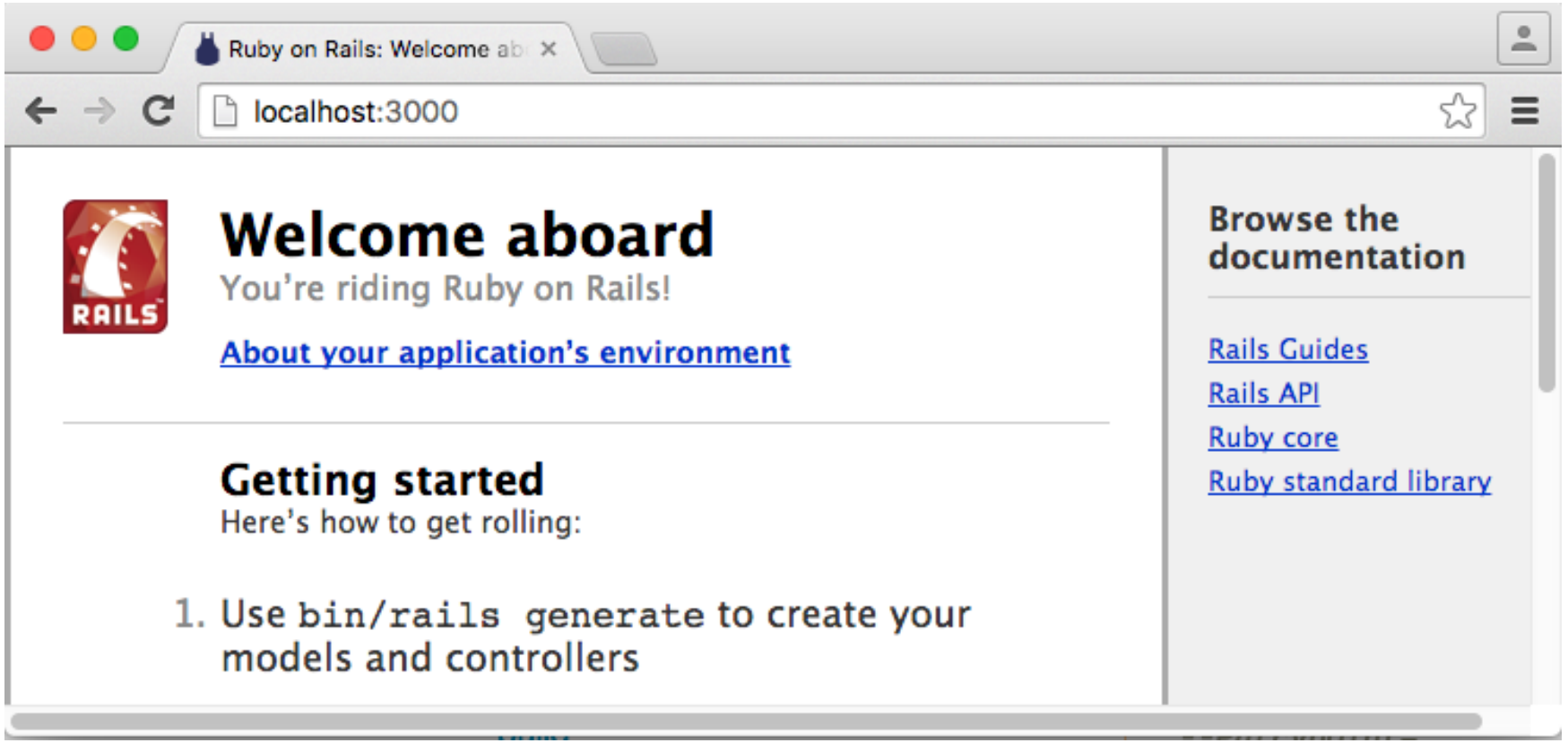


docker-compose up

```
david:docker-demo david.elner$ docker-compose up
Building web
Step 1 : FROM rails:4.2.6
 ---> 299e53ed9d2a
Step 2 : MAINTAINER David Elner <david@davidelner.com>
 ---> Using cache
```

```
Creating docker-demo-web
Attaching to docker-demo-web
docker-demo-web | [2016-06-16 20:14:27] INFO  WEBrick 1.3.1
docker-demo-web | [2016-06-16 20:14:27] INFO  ruby 2.3.1 (2016-04-26) [x86_64-linux]
docker-demo-web | [2016-06-16 20:14:27] INFO  WEBrick::HTTPServer#start: pid=1 port=3000
```

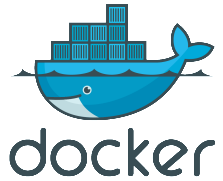
Start developing!



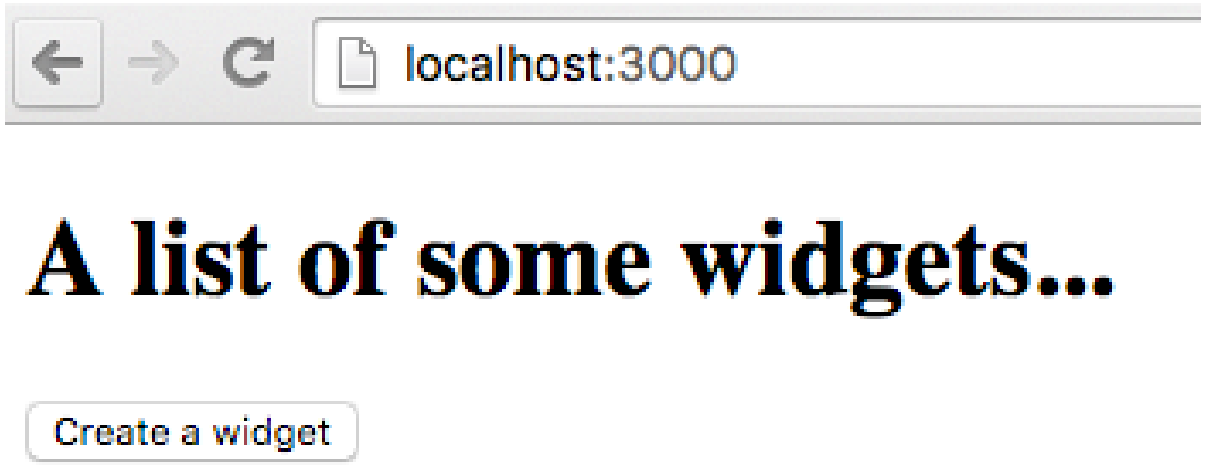
Adding another service

TURNING OUR APPLICATION INTO A SUITE

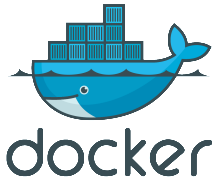
Adding a database



- We want our application to persist some data
  - So **we need to add a database service (e.g. Postgres)**
  - Let's add a Widget model, and a page to list & create widgets
  - You can click a button to generate one and save it

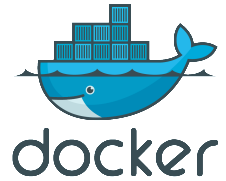


Adding Rails model…




```ruby
class Widget < ActiveRecord::Base
  def self.colors
    ['red','orange','green','teal','pink','yellow']
  end

  before_save :default_values
  def default_values
    self.color ||= self.class.colors.sample
  end
end

```

Adding Rails controller/view…

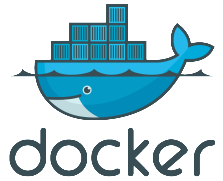



```ruby
class HomeController < ApplicationController
  def index
    @widgets = Widget.all
  end

  def create
    @widget = Widget.create
    redirect_to root_path
  end
end

```

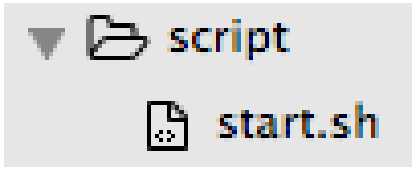
```erb
<h1>A list of some widgets...</h1>
<%= form_tag(create_path, method: "post") do %>
  <%= submit_tag("Create a widget") %>
<% end %>
<br/>
<% @widgets.each do |w| %>
  <h3 style="color: <%= w.color %>">Widget</h3>
<% end %>
<br/>

```

```ruby
Rails.application.routes.draw do
  root 'home#index'
  post '/create' => "home#create"
end

```

Adding migration support...

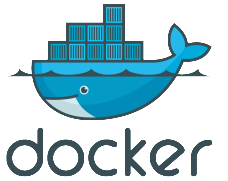


- We need to do more than `rails server` to start web now
  - Add a `**script/start.sh**` file
  - Setups up database, runs migrations, starts server




```
1   #!/bin/bash
2
3   echo "Running migrations..."
4   bundle exec rake db:migrate
5
6   echo "Cleaning tmp dir..."
7   rm -rf tmp/unicorn.pid
8   rm -rf tmp/pids/server.pid
9   bin/rake tmp:clear
10
11  echo "Starting Rails server..."
12  bundle exec rails server -b 0.0.0.0
```

Configure the environment…



- Our application and database will need to know the user/password
  - Setup some environment variable files.
  - Add them to .gitignore

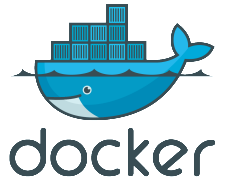
common.env

```
1  POSTGRES_PASSWORD=mysecretpassword
2  POSTGRES_USER=docker_demo
```

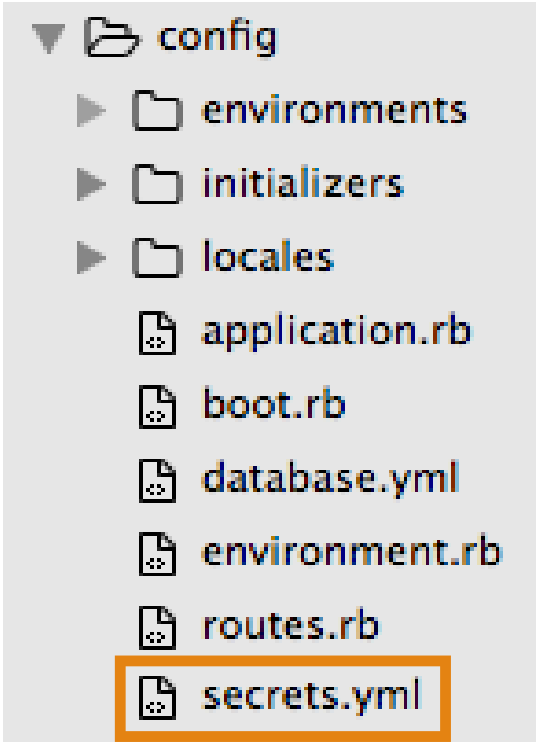
web.env

```
1  RAILS_ENV=development
2  SECRET_KEY_BASE=3305f4
```

Configure the environment…

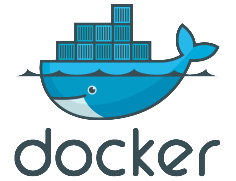


- Update the **secrets.yml** file to use environment variables

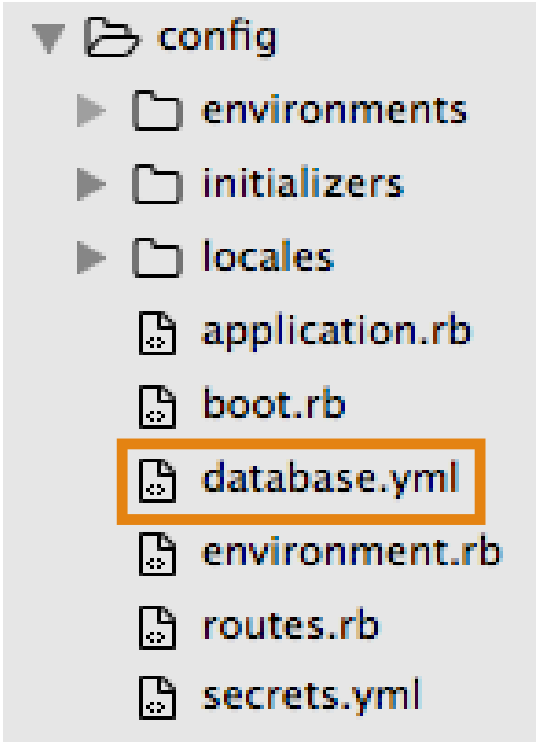



```
development:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>

test:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>

production:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>

```

Configure the environment…



- Update the **database.yml** file to use environment variables




```
1  default: &default
2    adapter: postgresql
3    encoding: unicode
4
5    host: db
6
7    username: <%= ENV['POSTGRES_USER'] %>
8    password: <%= ENV['POSTGRES_PASSWORD'] %>
9
10 development:
11   <<: *default
12   database: docker_demo_development
13
14 test:
15   <<: *default
16   database: docker_demo_test
17
18 production:
19   <<: *default
20   database: docker_demo_production
21
```

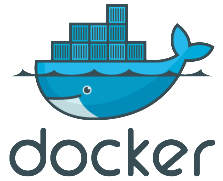
NOTICE:

Use of “**db**” as host.
This is a special name.

It matches the DB container name.

We’ll see this in next slide.

Adding a new service…

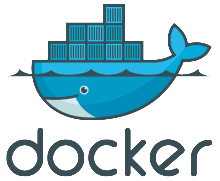


In our **docker-compose.yml** file, add the **db** definition:

Matches host from **database.yml**

```
16 ▼  db:
17        container_name: docker-demo-db
18        image: postgres
19 ▼      env_file:
20            - ./config/environments/development/common.env
21            - ./config/environments/development/db.env
```

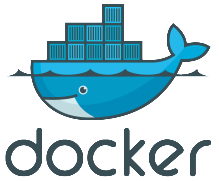
Link the containers...



In our **docker-compose.yml** file, modify the **web** definition:

```
1  version: '2'
2  services:
3    web:
4      container_name: docker-demo-web
5      build: .
6      volumes:
7        - .:/app
8      env_file:
9        - ./config/environments/development/common.env
10       - ./config/environments/development/web.env
11     command: /bin/bash ./script/start.sh
12     ports:
13       - "3000:3000"
14     depends_on:
15       - db
```

Start the database…



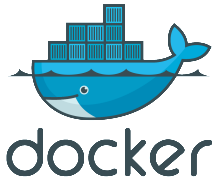
docker-compose up db

```
david:docker-demo david.elner$ docker-compose up db
Creating volume "dockerdemo_docker-demo-db-data" with default driver
Creating docker-demo-db
Attaching to docker-demo-db
docker-demo-db | The files belonging to this database system will be owned by user "postgres".
docker-demo-db | This user must also own the server process.
docker-demo-db |
docker-demo-db | The database cluster will be initialized with locale "en_US.utf8".
docker-demo-db | The default database encoding has accordingly been set to "UTF8".
docker-demo-db | The default text search configuration will be set to "english".
docker-demo-db |
docker-demo-db | Data page checksums are disabled.
```

Then create the database using Rails via:

docker-compose run web rake db:create

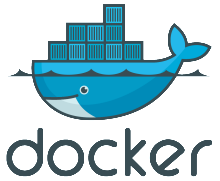
Start the suite…



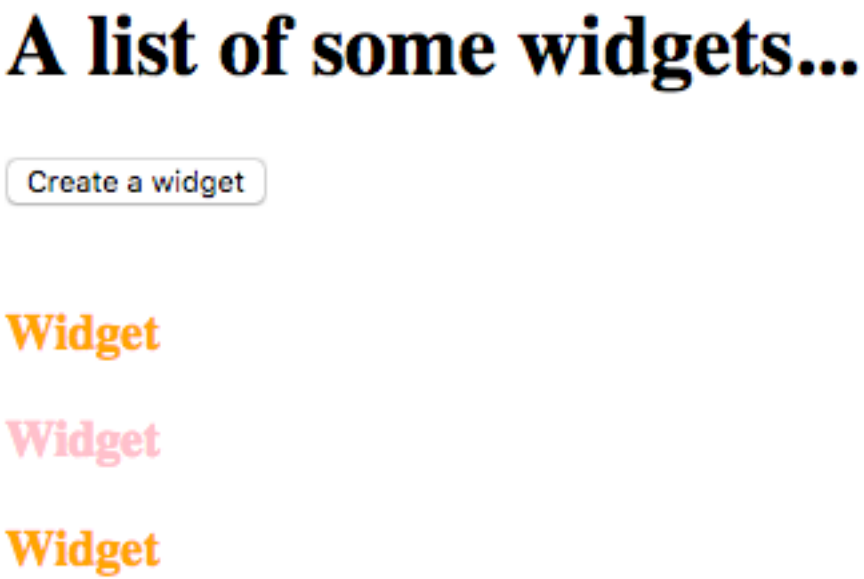
docker-compose up

```
docker-demo-db | LOG:  MultiXact member wraparound protections are now enabled
docker-demo-db | LOG:  autovacuum launcher started
docker-demo-db | LOG:  database system is ready to accept connections
docker-demo-web | == 20160617144000 CreateWidgets: migrating ====================================
docker-demo-web | -- create_table(:widgets)
docker-demo-web |    -> 0.0060s
docker-demo-web | == 20160617144000 CreateWidgets: migrated (0.0060s) ===========================
docker-demo-web |
docker-demo-web | Cleaning tmp dir...
docker-demo-web | Starting Rails server...
docker-demo-web | [2016-06-22 16:20:29] INFO  WEBrick 1.3.1
docker-demo-web | [2016-06-22 16:20:29] INFO  ruby 2.3.1 (2016-04-26) [x86_64-linux]
docker-demo-web | [2016-06-22 16:20:29] INFO  WEBrick::HTTPServer#start: pid=12 port=3000
```

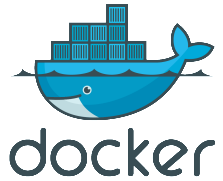
It works!



Clicking **Create Widget** saves a bunch of Widgets to the database.



So I'll just restart my application...

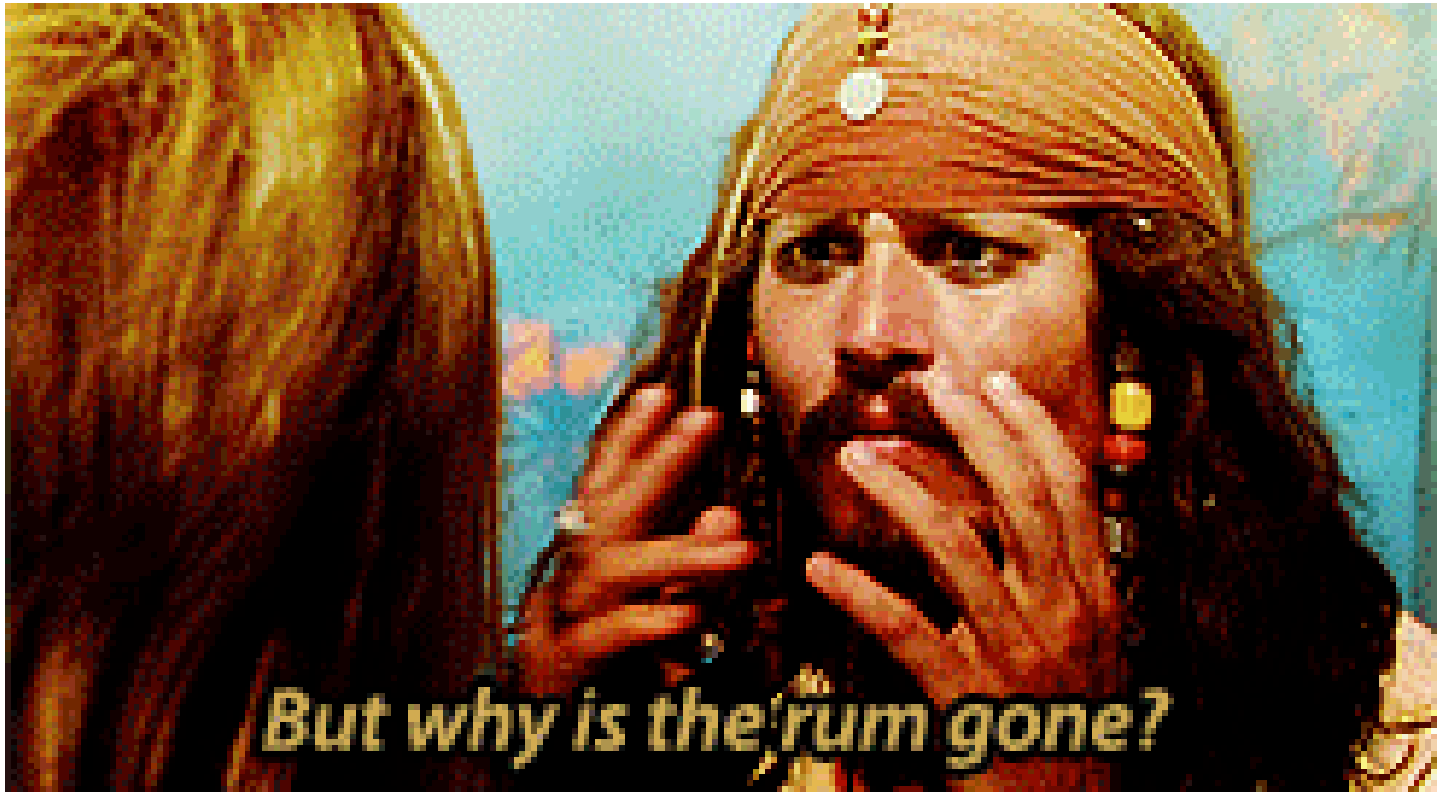


Revisiting the page after stopping the suite with **Ctrl+C** and running **docker-compose up** again…

A list of some widgets...

Create a widget

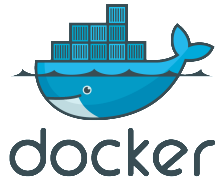
Where did all the data go?



Persisting data

HOW NOT TO LOSE EVERYTHING

Changes are ephemeral!

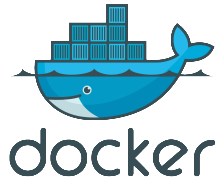


- Any changes you make to the container are temporary.
  - After stopping a container, you can **docker commit** to save them.
  - Otherwise restarting or killing a container will discard them.

- What about databases? How do I persist data between sessions?
  - Baking application data into your image with **docker commit** is frowned upon.
  - Instead, use **volumes**. They're directories you can mount onto containers.

Think USB thumb drive.

Using a volume…

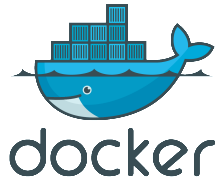


In our **docker-compose.yml** file, add **volumes**, and modify the **db** definition:

```
24  volumes:
25    docker-demo-db-data:
```

```
16▼   db:
17      container_name: docker-demo-db
18      image: postgres
19▼     env_file:
20        - ./config/environments/development/common.env
21        - ./config/environments/development/db.env
22      volumes:
23        - docker-demo-db-data:/var/lib/postgresql/data
```

Re-doing database setup…



docker-compose up db

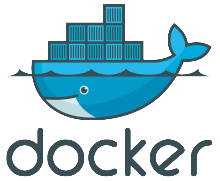
```
david:docker-demo david.elner$ docker-compose up db
Creating volume "dockerdemo_docker-demo-db-data" with default driver
Creating docker-demo-db
Attaching to docker-demo-db
docker-demo-db | The files belonging to this database system will be owned by user "postgres".
docker-demo-db | This user must also own the server process.
docker-demo-db |
docker-demo-db | The database cluster will be initialized with locale "en_US.utf8".
docker-demo-db | The default database encoding has accordingly been set to "UTF8".
docker-demo-db | The default text search configuration will be set to "english".
docker-demo-db |
docker-demo-db | Data page checksums are disabled.
```

docker-compose run web rake db:create

docker-compose up

```
docker-demo-db | LOG:  MultiXact member wraparound protections are now enabled
docker-demo-db | LOG:  autovacuum launcher started
docker-demo-db | LOG:  database system is ready to accept connections
docker-demo-web | == 20160617144000 CreateWidgets: migrating ======================================
docker-demo-web | -- create_table(:widgets)
docker-demo-web |    -> 0.0060s
docker-demo-web | == 20160617144000 CreateWidgets: migrated (0.0060s) =============================
docker-demo-web |
docker-demo-web | Cleaning tmp dir...
docker-demo-web | Starting Rails server...
docker-demo-web | [2016-06-22 16:20:29] INFO  WEBrick 1.3.1
docker-demo-web | [2016-06-22 16:20:29] INFO  ruby 2.3.1 (2016-04-26) [x86_64-linux]
docker-demo-web | [2016-06-22 16:20:29] INFO  WEBrick::HTTPServer#start: pid=12 port=3000
```

Adding some data...



Click “Create a widget”x3

A list of some widgets...

Create a widget

Widget

Widget

Widget

Ctrl+C & docker-compose up

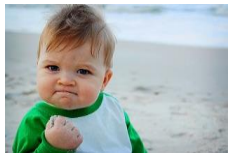
A list of some widgets...

Create a widget

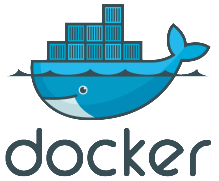
Widget

Widget

Widget



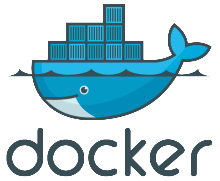
Success!



So what does it mean?

HOW DOCKER MAKES LIFE EASIER

Let’s revisit Brienne’s pitch…



- ✓ Can application setup be done in a fraction of the time?

  docker run *makes this fast.*

- ✓ Can application requirements be ‘versioned’ and shared easily?

  docker-compose.yml *defines everything our application needs.*

- ✓ Can we deploy new services without expert knowledge of that service?

  Just pull any image and start it in a container, to run them instantly.

- ✓ Can an entire suite of microservices be started locally with a single command?

  docker-compose up *makes this easy.*

Try it out for yourself!

https://github.com/blueapron/docker-demo

The end.

QUESTIONS?