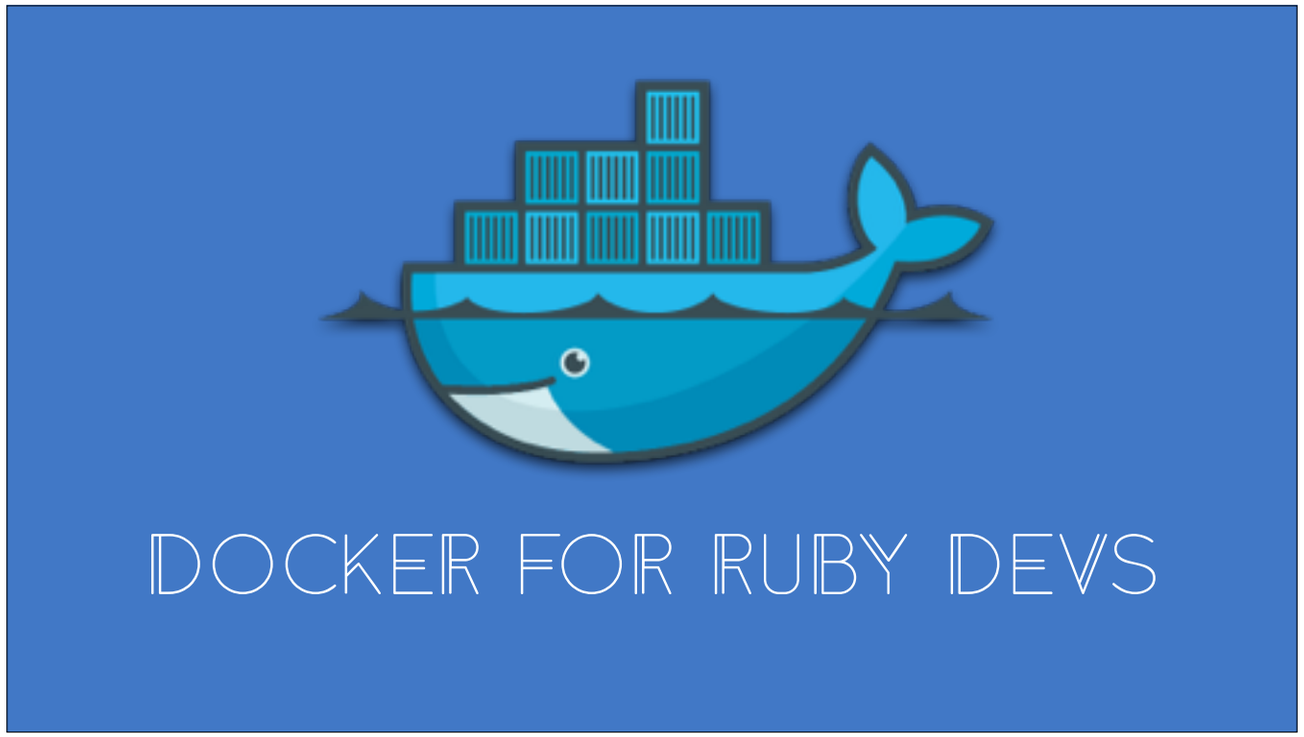
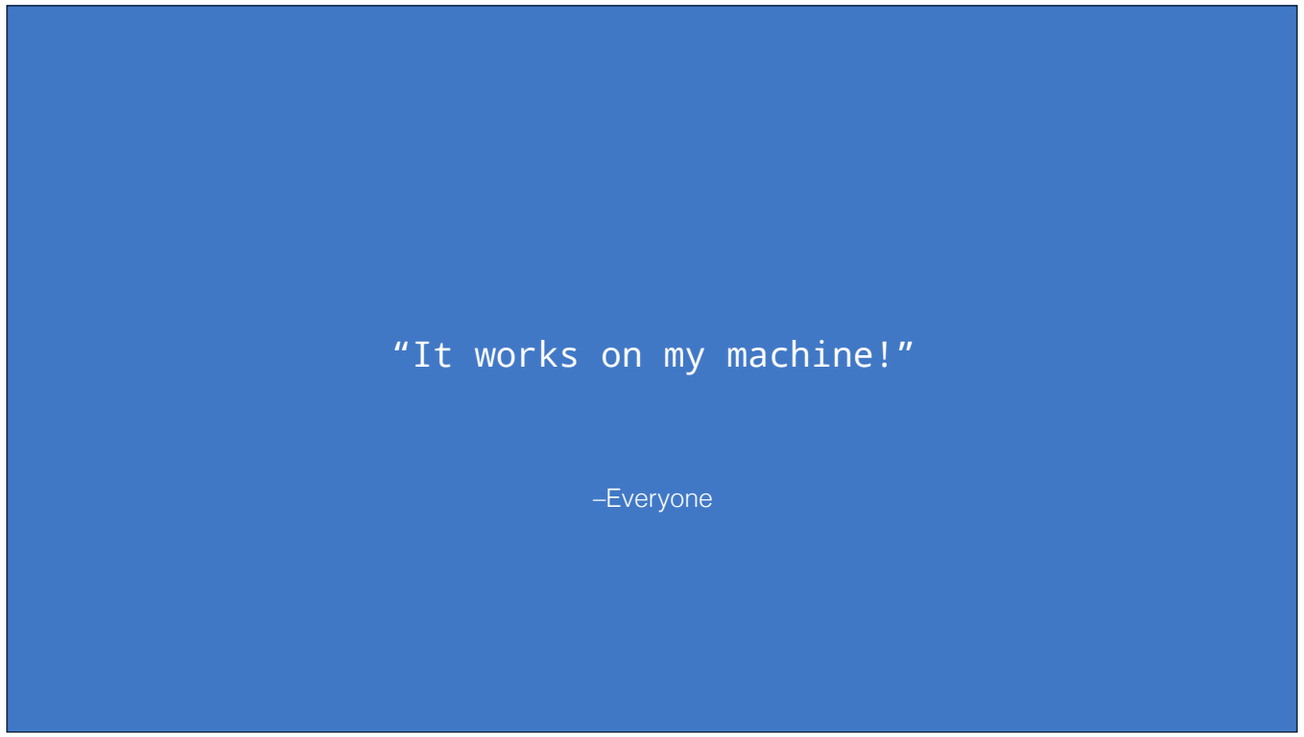


Probably most of you have heard about Docker by this point.
It's the new buzzword that gets thrown around but most people don't really know what it is or what it does.
Does anybody here know what docker is?
Docker isn't one product, it isn't something new. It's a wrapper around a lot of standard linux utilities and tools, but it makes those tools easier to use.

01

THE PROBLEM



Now I will tell you a true story that happened to me while working on a project.

So we worked on a complete book printing studio solution and we had to spit out images in CMYK colorspace.
Just to clarify for those who don’t know what CMYK is.
CMYK stands for Cyan, Magenta, Yellow, Black and it’s the super colorspace of RGB which means that it has much more colors and it’s better suited for print as printers have exactly those 4 colors in them.

So after a days work I implemented a bunch of logic that did some crazy magic to reduce multiple page elements in to one single CMYK image.
I pushed my code to the remote and went home.
Somewhere around 7pm my manager sent me a message that something was wrong with the server, that it was returning images in weird colors.
So naturally I downloaded an image and the colors were all off, it was grainy and looked horrible.
I checked my local commit hash with the one on the server and to my surprise they were the same!
Ok, I figured something was wrong with the version of ImageMagick on the server so I checked my local version with the server’s and again they were the same!
Well, it was late and I decided to sleep about it.
The next day a college of mine, who was working on the fronted for the app, comes to my desk asks how I could commit broken code to the master branch.
Turns out the image processor spat out black & white images on his machine! Again we all had the same version of ImageMagic. What the hell was happening?

Perhaps somebody knows by now what went wrong?

Well, the server had all the right library versions but they weren’t compiled with the right flags and external libraries.

THE PROBLEM

- Different libraries
- Libraries have different bugs on different OS
- Different versions of libraries
- Upgrading / downgrading breaks other libraries

02 SOLUTION

It would be great if we could somehow create an environment which would be similar to the server and which we could all share and develop on.

The first thing that came to my mind was a VM.
There are many VM solutions in existence, but to me non of them ever felt right.
First off, the better VM solutions are quite expensive and work only on Mac.
They are clunky, often slow, eat up memory for no apparent reason and not to say that you need at least a GB of RAM to start one up.
Passing data and sharing resources can often be a pain…
They add an extra OS on top of my existing OS.
And all except vagrant are unpractical for development as you have to manage starting and stopping the VM by your self.
Not to mention mounting virtual disks, setting up the environment for each project.
Too much work, to little of a result.

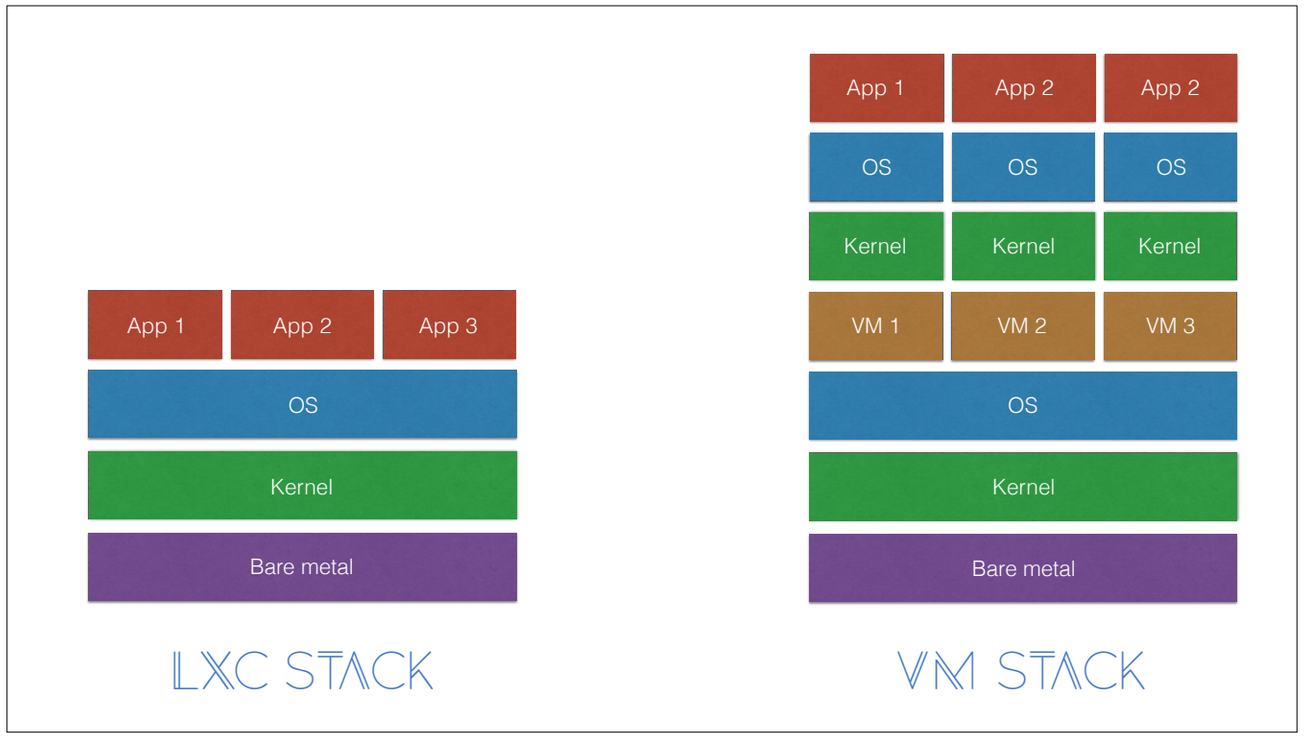
Surely there has to be a better way?

SOLUTION

Then I stubbled upon LXC
LXC stands for Linux containers.
First off, they are LINUX containers. I will mention how Mac handles this problem later.

LXC offers complete user space isolation.
That means that each container thinks it’s the soul user of the OS.
It shares every resource with the host OS (including the Kernel).

That would be the optimal solution!
It has nearly no memory overhead, it shares available resources and it enables us to craft an environment and to share it.



It’s really important to note that LXC is not a VM!

In LXC all containers or apps run directly on top of the host OS and share their resources as if they were processes in the host OS but enable you to install different libraries and apps as if the host OS didn’t exist.
In a VM stack you create separate installations of another OS with their own separate kernels, libraries and apps.

The problem of LXC is that of sharability.
To share an LXC you would have to create a script that creates a container and installs all the needed libraries in it.

03 DOCKER

Docker to the rescue!

Docker is a wrapper around LXC that provides much needed syntactic sugar.
It enables you to write simple cookbooks called dockerfiles that build your container for you in seconds.

DOCKER

Dockerfile — Raw

```
1  FROM rails:4.2.4
2  MAINTAINER Stanko Krtalic Rusendic <stanko.krtalic@gmail.com>
3  RUN mkdir -p /var/app
4  COPY Gemfile /var/app/Gemfile
5  WORKDIR /var/app
6  RUN bundle install
7  CMD rails s -b 0.0.0.0
```

```
$ cd local/project/path
$ docker build -t my_image_name .
$ docker run -v local/project/path:/var/app -p 3000:3000 my_image_name
```

This is an example of a simple Dockerfile.
Docker files live in the root of your project.

Then we run `docker build` it builds a LXC container for us and install everything we need to start working on a Rails project.
When we run `docker run` it starts up the container.

DOCKER

Dockerfile — Raw

```
1  FROM rails:4.2.4
2  MAINTAINER Stanko Krtalic Rusendic <stanko.krtalic@gmail.com>
3  RUN mkdir -p /var/app
4  COPY Gemfile /var/app/Gemfile
5  WORKDIR /var/app
6  RUN bundle install
7  CMD rails s -b 0.0.0.0
```

```
$ cd local/project/path
$ docker build -t my_image_name .
$ docker run -v local/project/path:/var/app -p 3000:3000 my_image_name
```

The `-t` gives the container a name
The `.` is really important! It specifies to build the local dockerfile
The `-v` option mounts the project's root as a virtual drive to the container's `var/app`
The `-p` option exposes the container's port 3000 to the host's port 3000

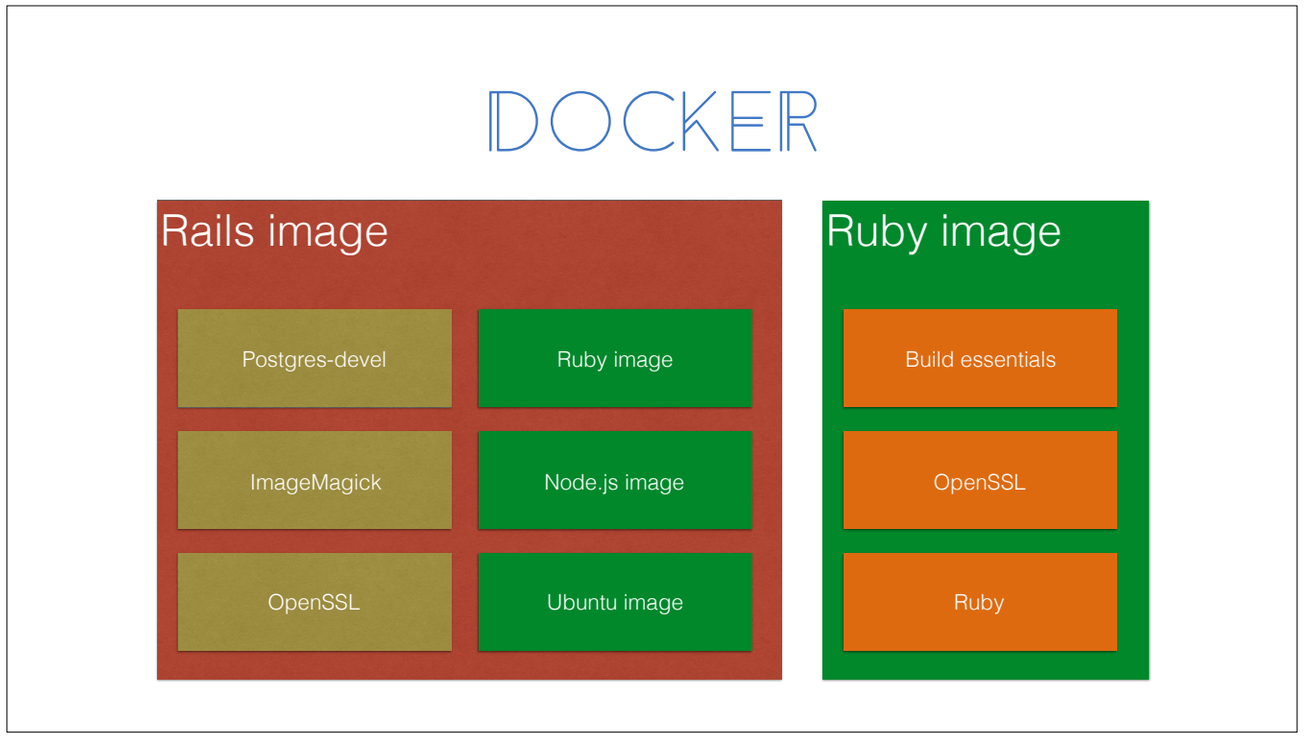
DOCKER

Dockerfile — Raw

```
1  FROM rails:4.2.4
2  MAINTAINER Stanko Krtalic Rusendic <stanko.krtalic@gmail.com>
3  RUN mkdir -p /var/app
4  COPY Gemfile /var/app/Gemfile
5  WORKDIR /var/app
6  RUN bundle install
7  CMD rails s -b 0.0.0.0
```

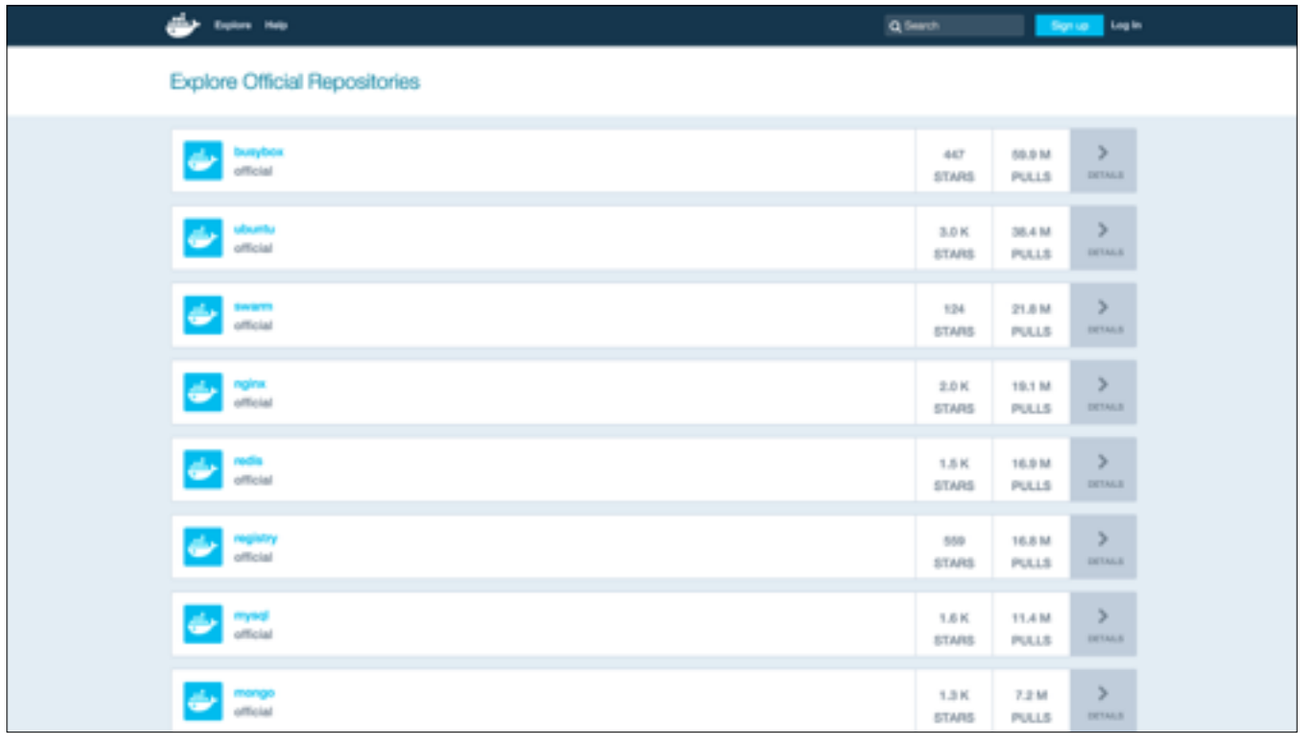
```
$ cd local/project/path
$ docker build -t my_image_name .
$ docker run -v local/project/path:/var/app -p 3000:3000 my_image_name
```

Ok, but how did it know what we need to run Rails? What did it install?
Well we said to it in the docker file that it should use the Rails 4.2.4 image to create the container.
What are images?



In the docker world we differentiate two types of images, mixin images and standalone images.

Mixing images are used to create other images.
As shown above, to create the Rails image we use the Ruby image which gives the Rails image a working installation of Ruby.
Not that the Rails image won’t spawn a container for each of it’s sub images but they will all be merged in to one image and will only spawn one container.



Images are incredibly cool!

You can think of them as Ruby classes and each container created from a single image is an instance of the image.

Each is able to create a complete environment in minutes without anybody administering it.
What's even cooler and what differentiates it from server and VM provisioning tools like Ansible, Puppet and Chef is a huge library of pre-made images most of which are 'official' which means they are configured and maintained by the creator of the software and optimised for it. They can be combined, used as dependencies and extended to fit your specific need, much like gems in Ruby.

Go to hub.docker.com and search for something, you will even find Minecraft in there.

DOCKER

15 lines (11 sloc) | 426 Bytes — Raw | Blame | History

```
FROM ubuntu
MAINTAINER Kimbro Staken

RUN apt-get install -y software-properties-common python
RUN add-apt-repository ppa:chris-lea/node.js
RUN echo "deb http://us.archive.ubuntu.com/ubuntu/ precise universe" >> /etc/apt/sources.list
RUN apt-get update
RUN apt-get install -y nodejs
#RUN apt-get install -y nodejs=0.6.12~dfsg1-1ubuntu1
RUN mkdir /var/www

ADD app.js /var/www/app.js

CMD ["/usr/bin/node", "/var/www/app.js"]
```

Images are created from docker files.
Or rather said, a docker file gets built to an image.

This is a simple dockerfile that creates a Node.js server.
First we say that we want to use Ubuntu as our base container on top of which we are going to build our container.
Then we install node and all it’s dependencies.
And finally run app.js.

Docker files are fine and great. but they leave plenty of room for fuckups as you have to do white a lot of work to get a simple application up and running.

DOCKER

docker-compose.yml — Raw

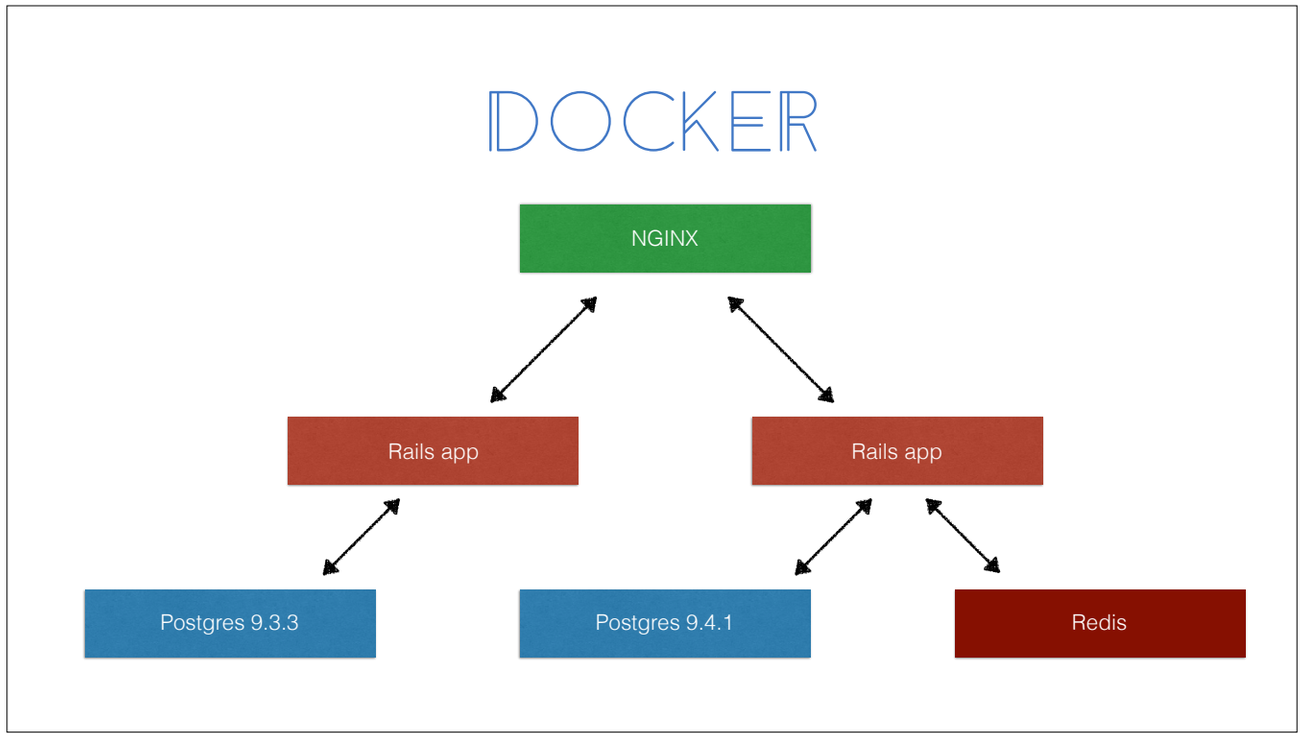
```
1  db:
2    image: postgres
3
4  web:
5    build: .
6    ports:
7      - 3000:3000'
8    volumes:
9      - 'local/project/path:/var/app'
10   command: rails s -b '0.0.0.0'
11   links:
12     - db
```

```
$ docker-compose up
```

```
$ docker-compose run bundle exec rake db:create db:migrate
```

Docker compose, formerly fig, is a tool that comes with the docker toolbox.
It enables you to create templates for applications and script application setup.
As you can see from the example it can also start and link containers together.

Arbitrary commands can be executed within the container using `docker-compose run`



In the previous example we linked the `db` and `web` containers together.
What does that mean?

Containers can be linked together so you don’t have to build monolithic containers that host multiple services in them.
This makes it possible to have several versions of the same application on the same machine.
It also enables your apps to use other apps as services which makes horizontal scaling a breeze.

This is where docker’s mantra resides.
Build many small services and link them together to build a greater app!

04 DEPLOYMENT

That would be most of the advantages while developing an app.
I go through which advantages you get if you run Docker on your servers as well.

ADVANTAGES

- 100% identical configuration
- Apps are better optimised
- Horizontal scaling
- Git-like filesystem
- Portability

There are four big advantages of using Docker on your servers.

First off you can be sure that your code will run exactly as expected, at least hardware wise :)

Apps can be better optimised because you can configure the environment for each individual app independently of other apps and services.

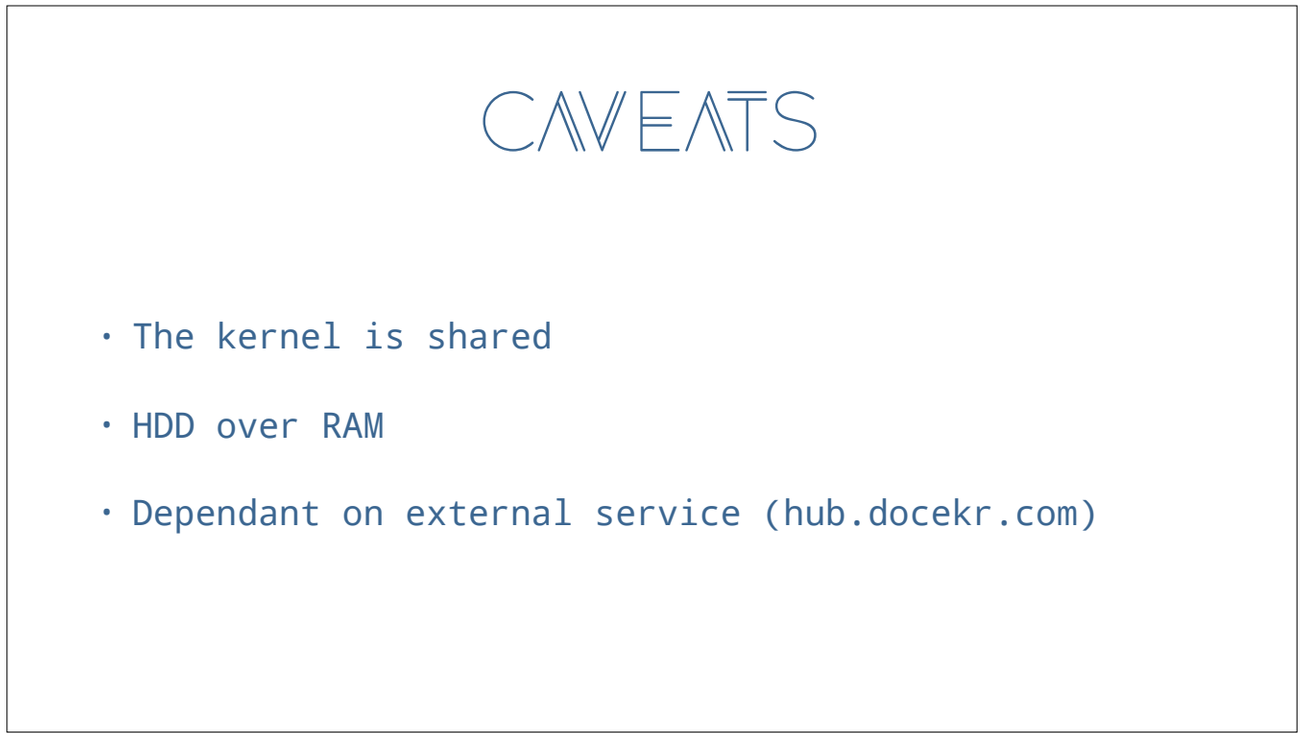
Horizontal scaling becomes a breeze! Need a new instance of the app? Just run `docker-compose up` and viola a new app has been spawned!

Something that has proven useful in production is docker’s git-like filesystem. You can commit, export and revert your container to any previous state.

The greatest advantage is surely portability. Suddenly your app can run on any server, it’s completely undefended of the hardware and other services because it uses everything for the kernel level up as a service.

05

CONCLUSION



There are a few caveats.

Something that makes Docker so snappy is the saved kernel. Well turns out that’s a double edged sword. Some apps require a specific kernel version, but the kernel is the only thing that connects all the components together.

Docker has a negelable memory overhead of 1MB, but because everything is kept separate a HDD space concern exists. Every app has a copy of all libraries and executables needed for it to run which chews up HDD space.

Docker is dependant on the Docker Hub. That can be a weak point if it ever goes down :S

THANKS FOR LISTENING

stanko.krtalic@gmail.com
@monorkin
https://github.com/Stankec