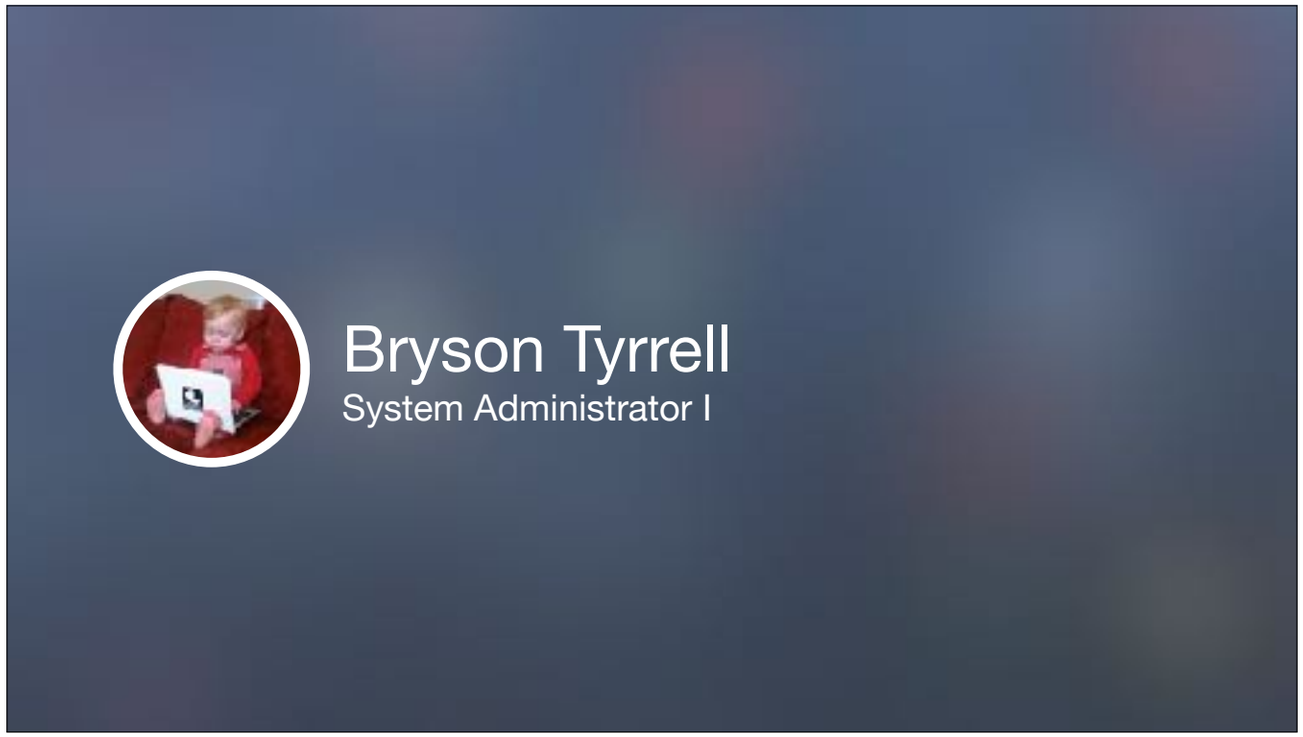

How Docker Compose Changed My Life

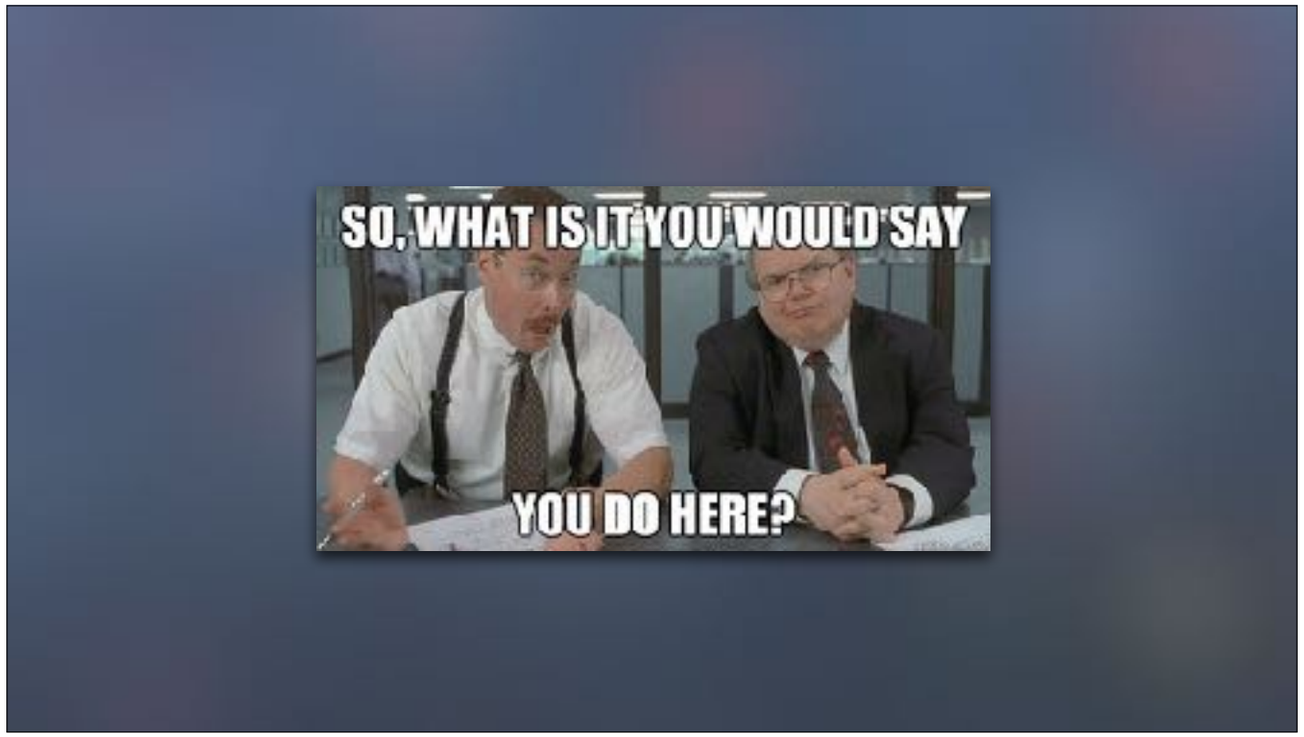
I’m Bryson Tyrrell, a Desktop Services Specialist with Jamf’s IT department.

Or at least I was…



I have, since I submitted this presentation, transitioned from the IT team to the Cloud & Delivery team which is responsible for Jamf Cloud.

I’m here to talk about how Docker Compose changed my life, but for that idea to make any sense it might help to know what is it I do at Jamf.

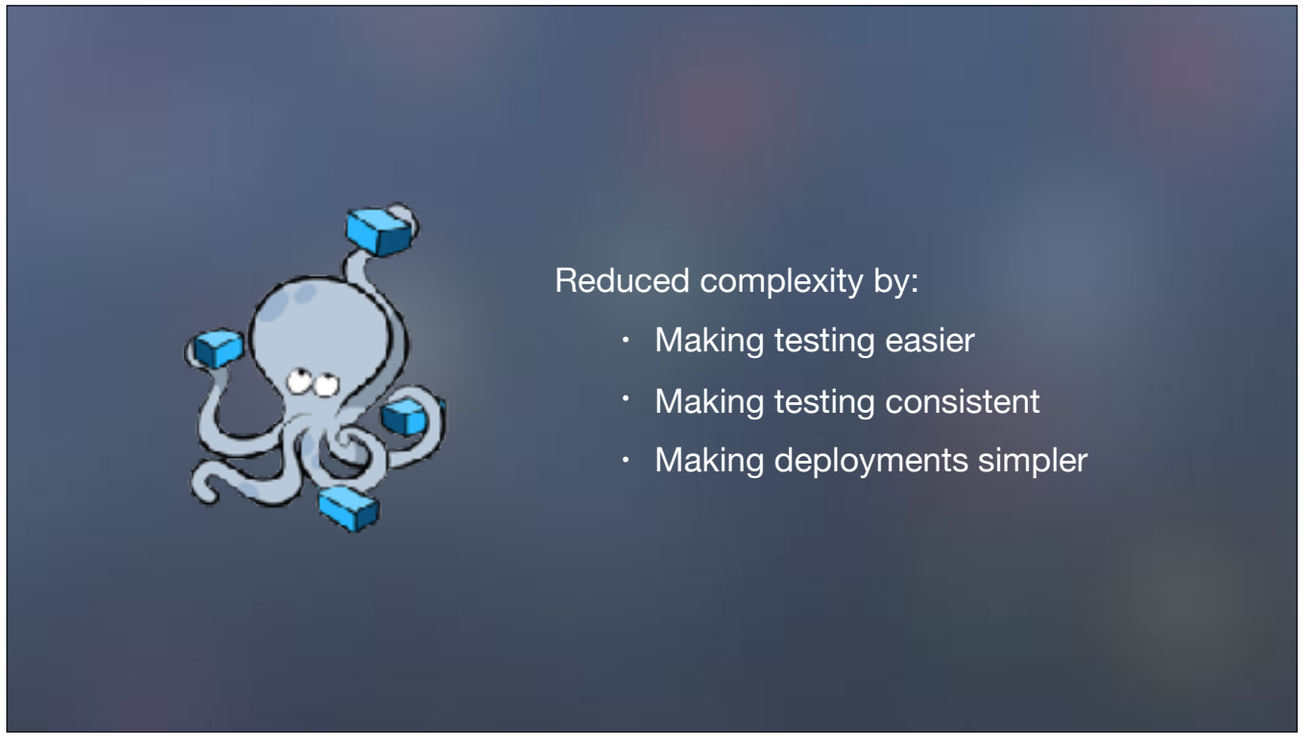


Well, in the last year I spent in IT my focus shifted to developing small internal integrations and services. These included things like our HipChat plugin to Jamf Pro, a photo server for demoing Apple School Manager and a DNS registration service.

As a System Administrator I am right now primarily focused on building new automation infrastructure for Jamf Cloud.

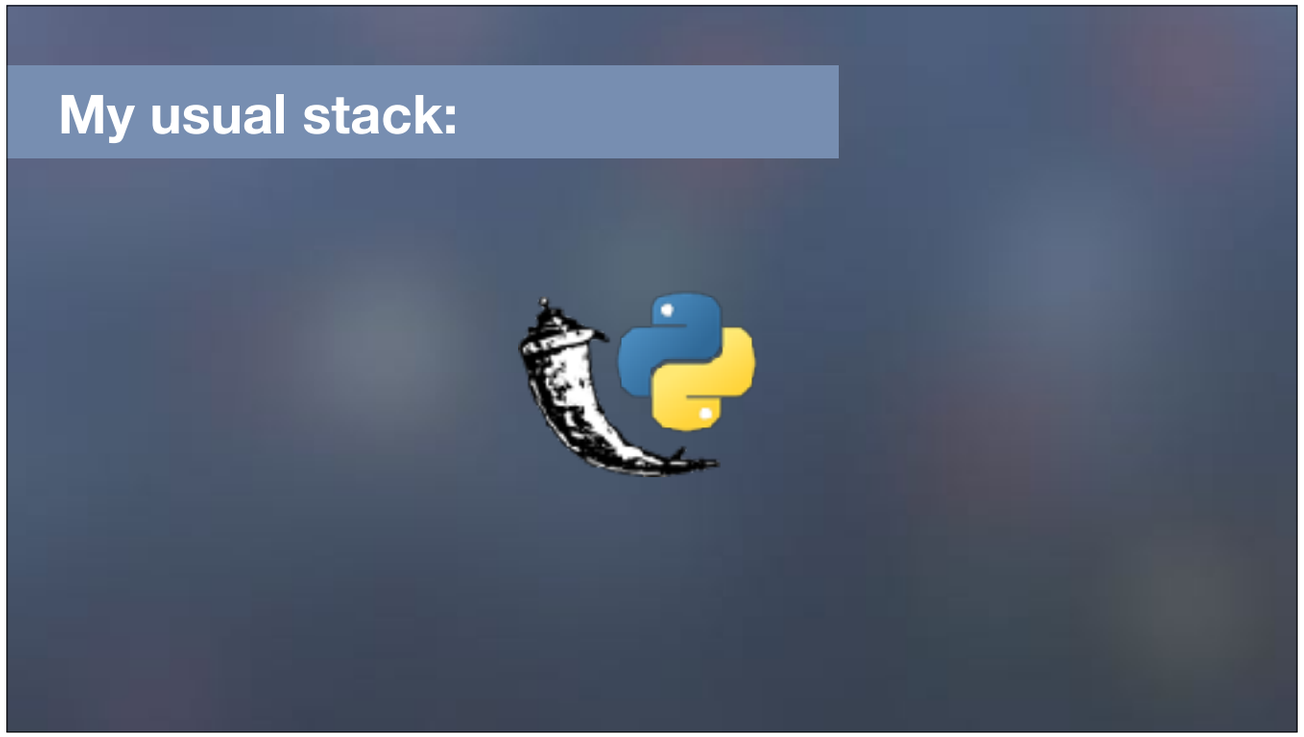
For all of this, I am writing web applications and services using Python.

So, how does Docker Compose fit in.



Docker Compose ended reducing complexity for me during the development process by:

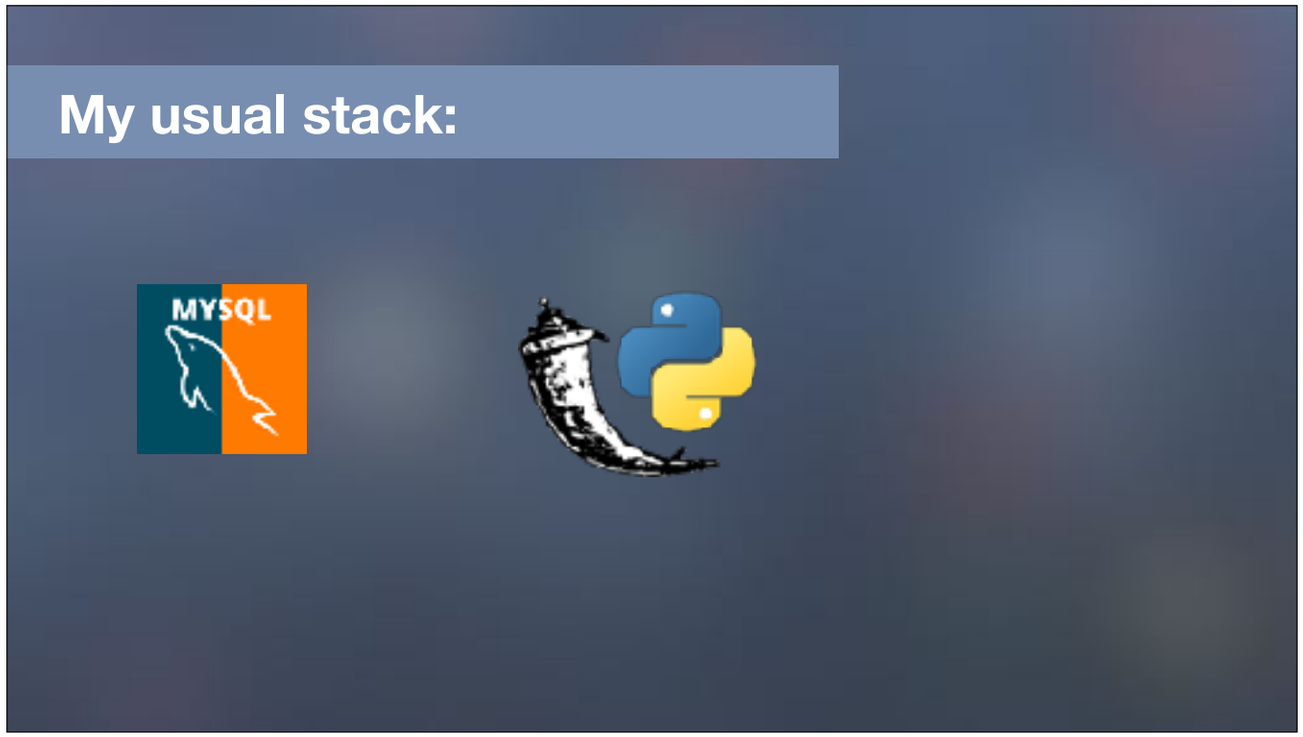
Making testing easier
Making testing consistent
Making deployments simpler



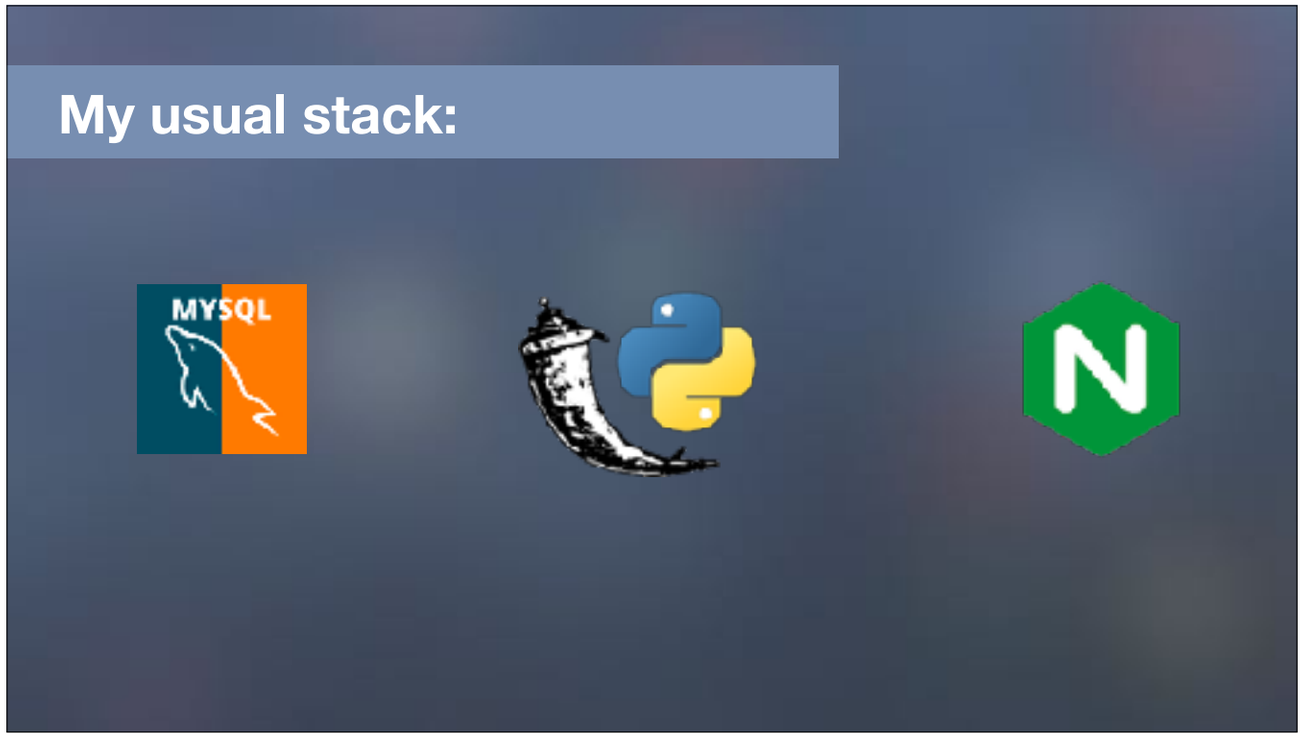
Let’s get right into how it did that.

As I said, all of my web applications are written in Python, and I have a pretty standard stack that I use.

At the core is Flask: a microframework for web applications. It’s simple, lightweight, very flexible, and you build everything you want on top of it instead of having everything together in one massive package.

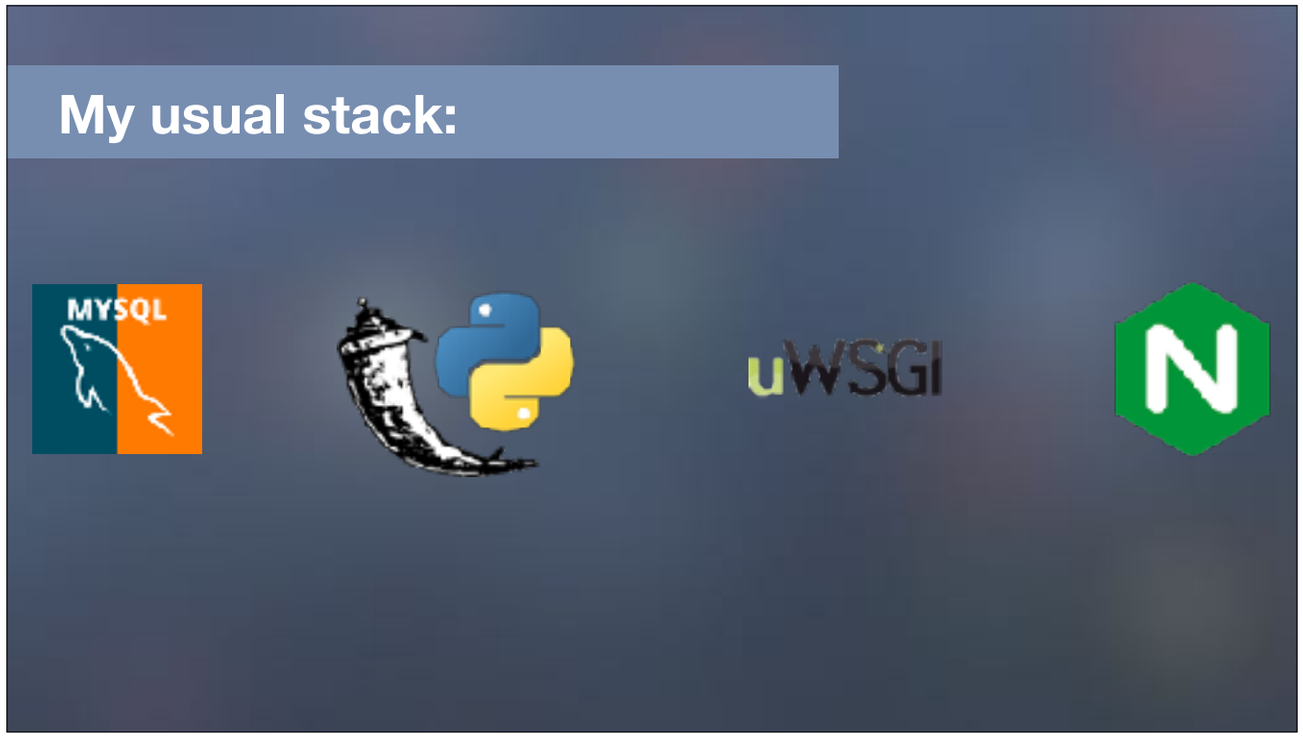


For the backend, I use MySQL databases for data storage. Nothing locks me into using MySQL, but most everyone else at Jamf knows it well enough.



On the front end I need a web server component to handle inbound requests and pass them to the application. Flask does contain a built-in server so you can start it up, but that's only meant for use during development when you want to spin it up and test things as you go.

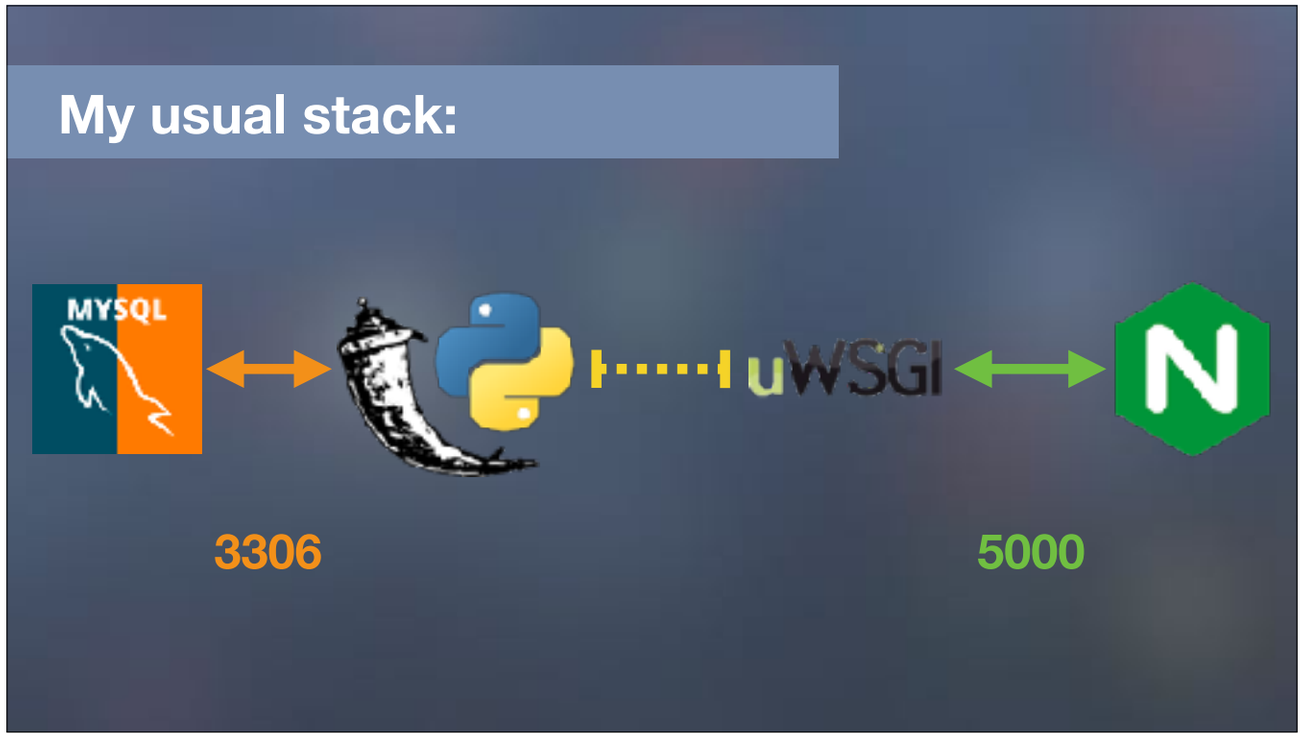
In my deployments I use Nginx for a lot of the same reasons I use Flask: it's simple, lightweight, and powerful.



In order for the web server to communicate with the application, there is another component that comes into play to provide an interface to executing the code. Flask is an implementation of a standard called WSGI (or Web Server Gateway Interface). A WSGI server accepts requests from a web server and calls the appropriate Flask code.

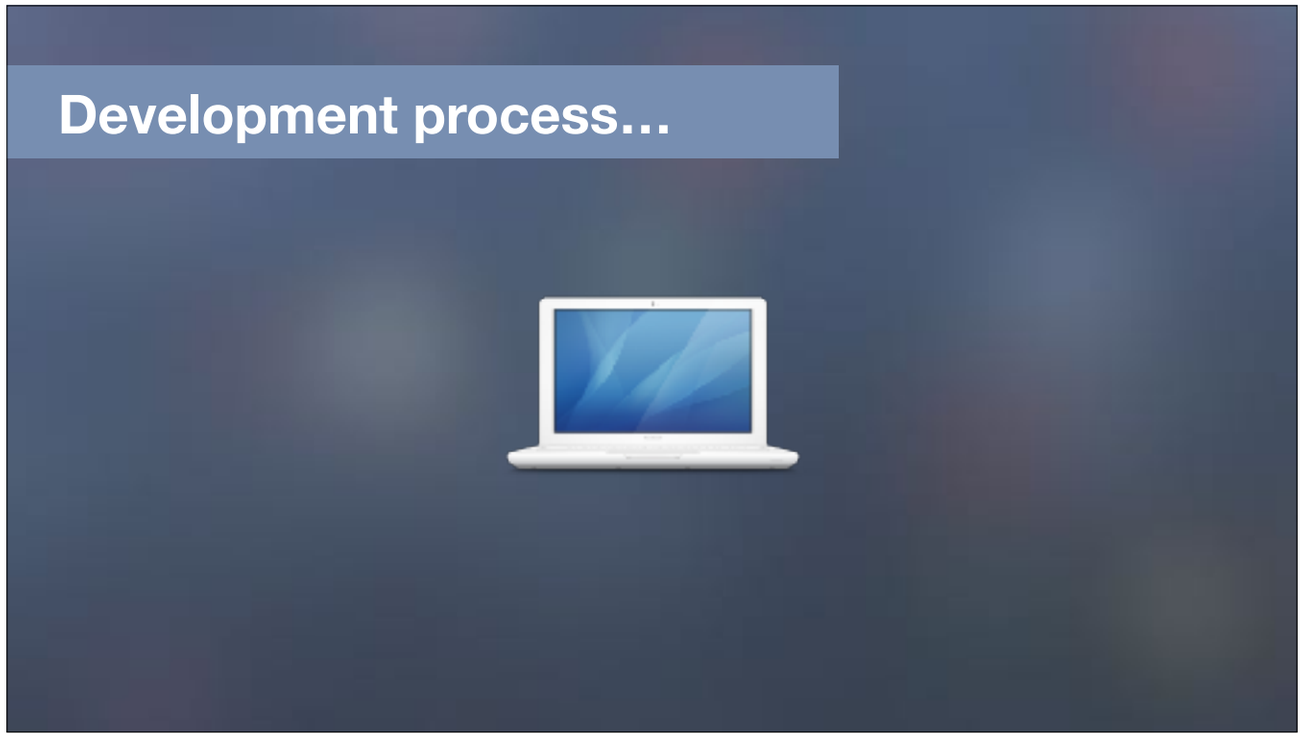
As it’s a standard. there are a lot of options out there for the WSGI server itself and they all have their strengths. I use the one named uWSGI for the most part.

See PEP 3333 for more

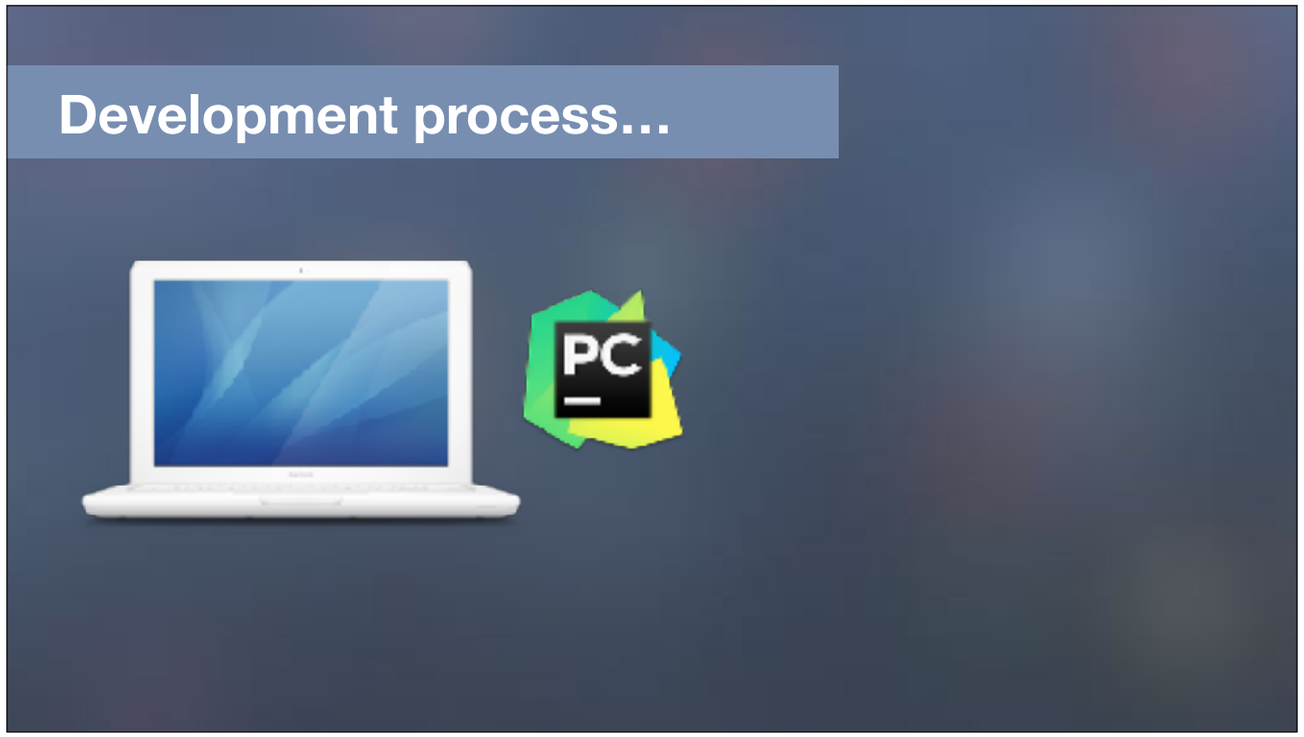


All the components of this stack communicate with each other.

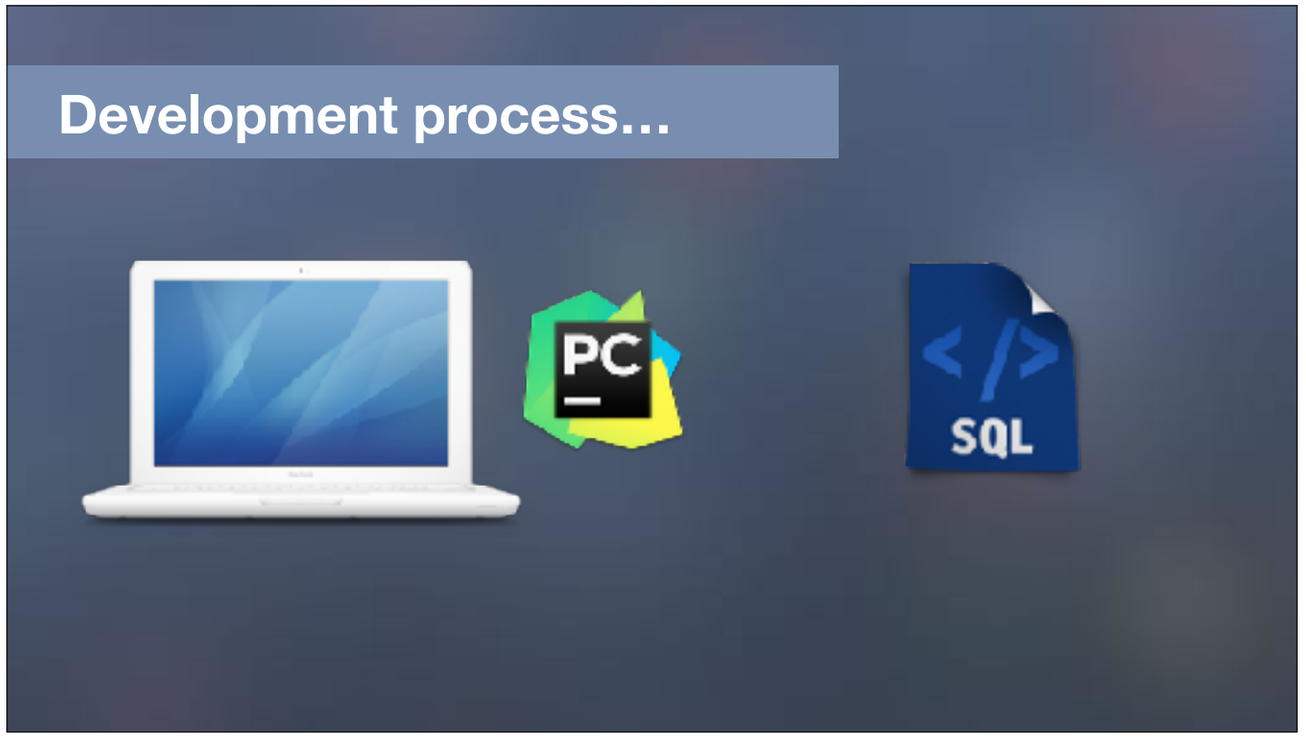
Requests come into Nginx over HTTPS and port 443.
Those requests are proxied to uWSGI which is running on port 5000.
uWSGI calls the relevant code in my Flask application.
The application makes calls to the MySQL database over port 3306.



Let’s take a look at how I used to go about the development process. Naturally I do all my work on top of the line hardware that Jamf provides.

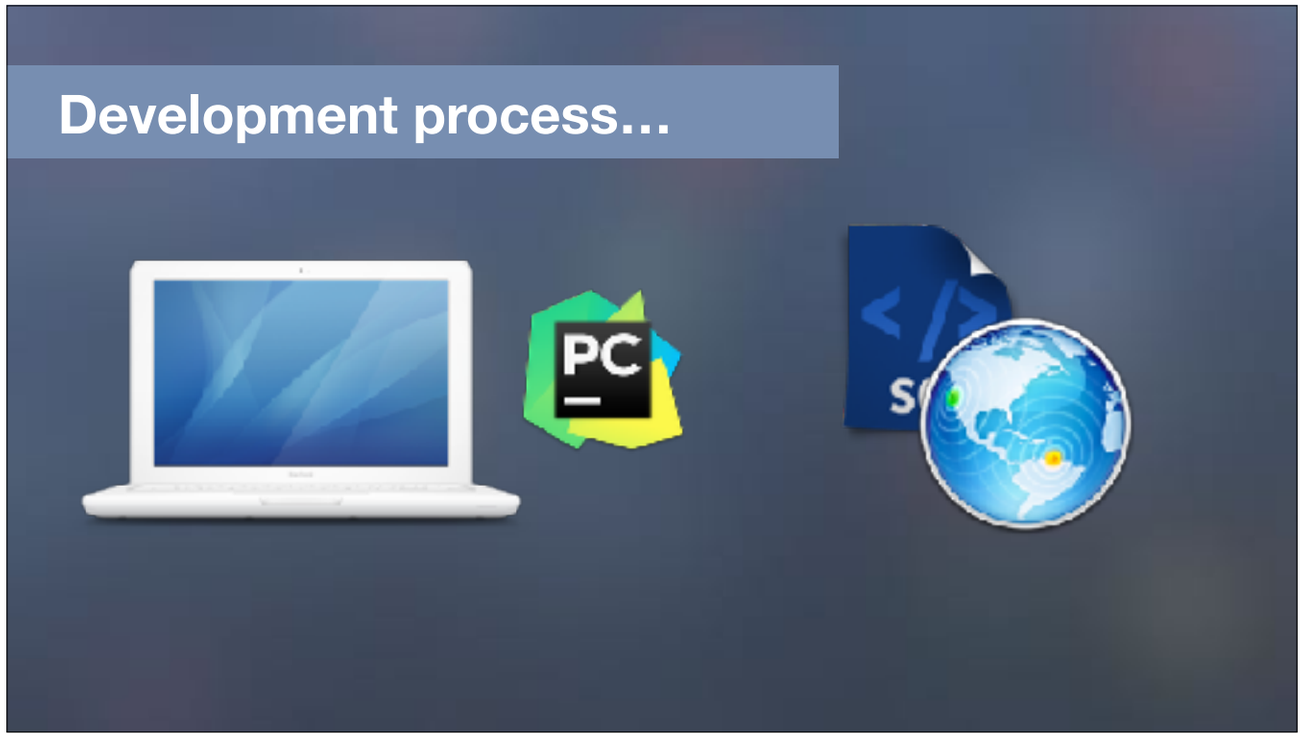


PyCharm is my IDE of choice. I also use Visual Studio Code for a lot of one-off files, especially if they’re in another language, but all of my project work is done here. IDEs make life so much easier with features like code completion, refactoring, and degbuggers so I can run my code and step through it.



Now, when I was doing that local testing I would set the configuration of my Flask app to use a local SQLLite file for the database. I don't like installing things like databases on my Mac. My Mac isn't a server and shouldn't be running those kinds of services. It was easier just to fire it up with a local file.

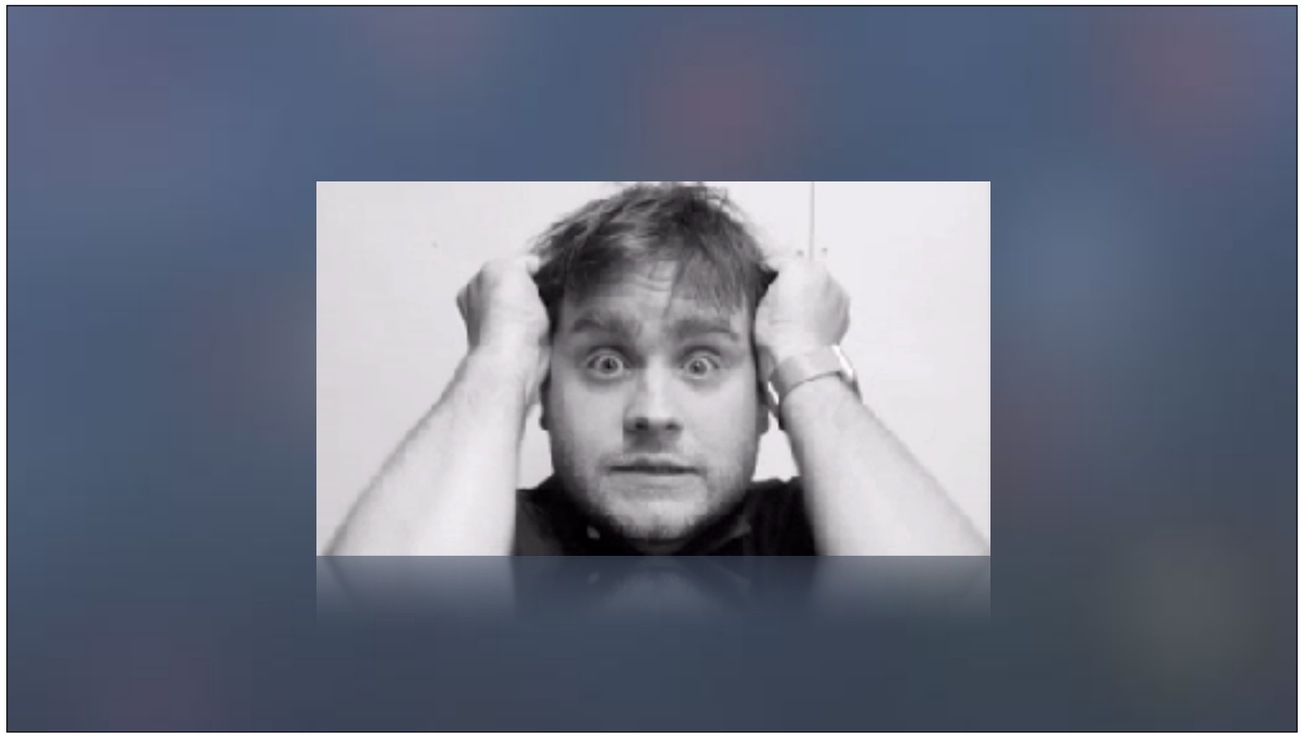
That has a lot of drawbacks. The biggest one is that I'm not testing against a running MySQL server. I could have bugs and issues with my database model definitions I don't know about until I actually try to connect to a running instance down the line. Maybe I didn't set an attribute or define the size or default of a column the way MySQL expects. What if I need to modify MySQL's configuration to account for the size of the data blobs that I'm writing back? At this point in the development process I'm oblivious to those potential problems.



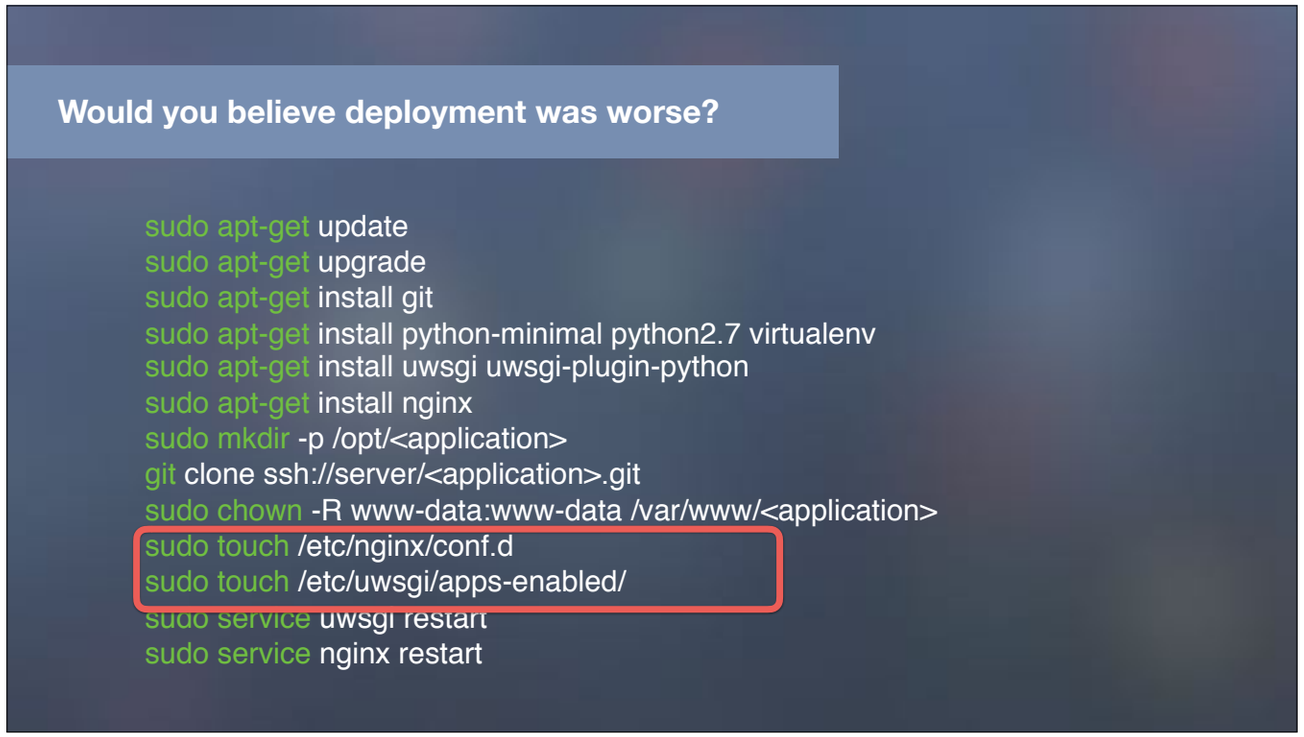
The next one it pretty apparent. I'm using the built in Flask development server to run the app. It's not being run by a WSGI server, and I'm not having a web server handling the requests I'm making while I test. There is always a level of configuration that needs to happen with the web server from one application to another.

Files are a great example. Large uploads and downloads work pretty well when doing them against the dev server. Nginx though is going to need to be configured for the max_body_size and the location of the static directory. The dev server and the actual web server behave differently, which isn't surprising.

With all these differences between my local testing and an actual deployment, when it got to the deployment all the extra work just led to…

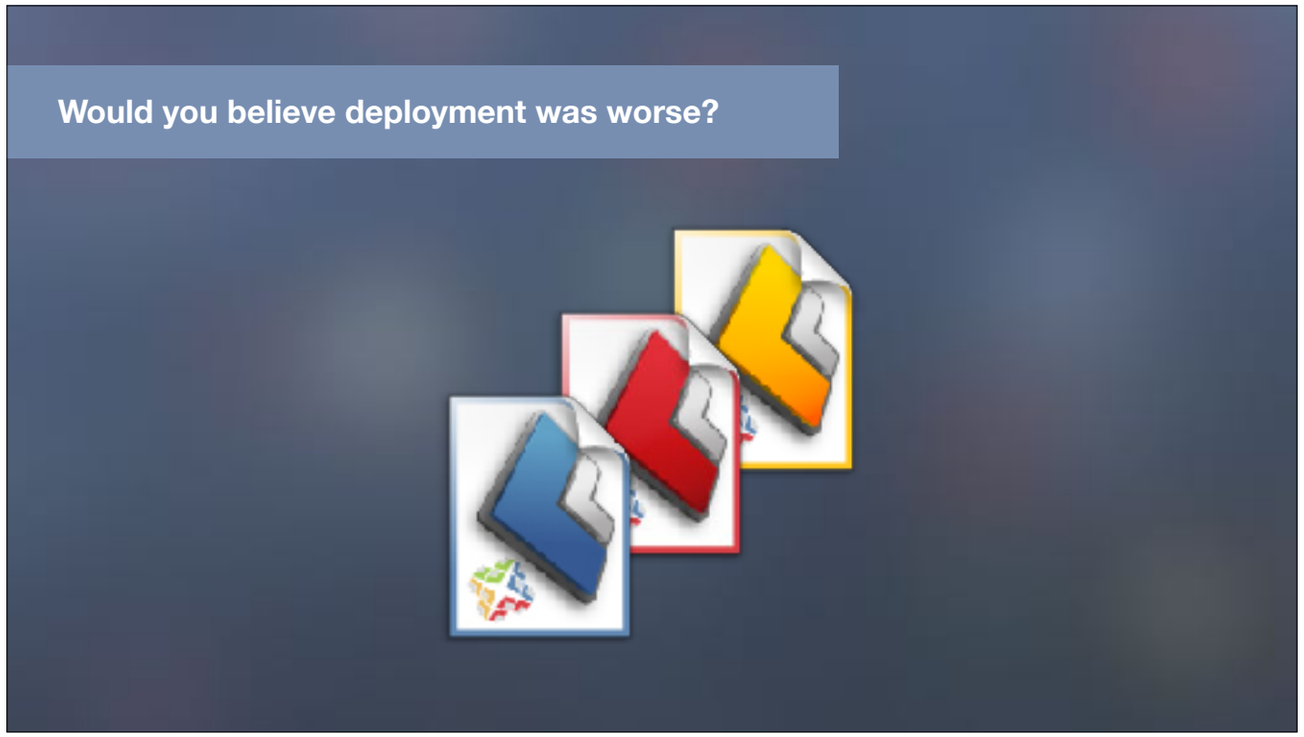


Nope. This clearly wasn’t working for me when it came to testing during development because I still had more deployment work waiting for me after I wrapped up the code.



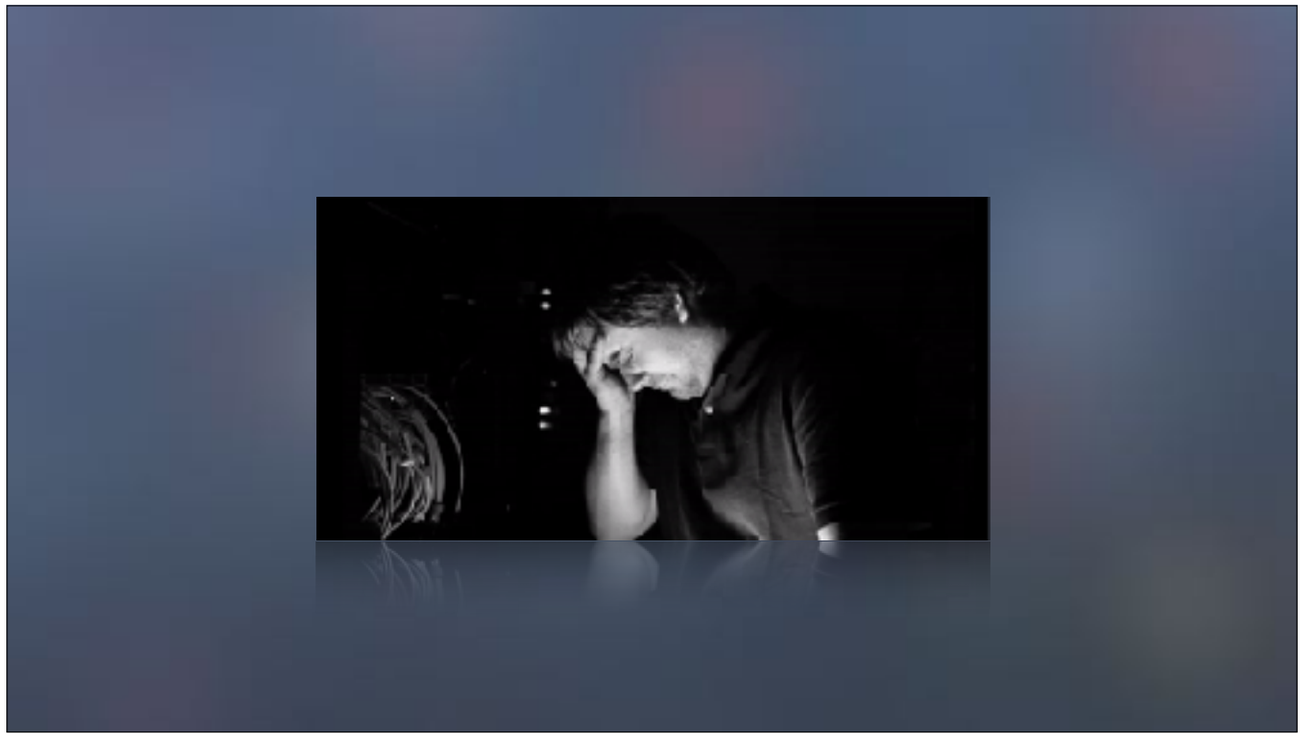
And when it came time to do a deployment, I needed a VM running Ubuntu 16.04 and I had to prep it to be able to run my code. That involves doing updates, installing system packages, install additional software like the MySQL, WSGI, and web servers, pull me code from the repository, load up the WSGI and Nginx configuration files, and start it.

And if I have to update the servers or the application code I have to repeat most of these steps.



Because my package requirements and configurations change between different applications, I needed to have a setup document, a one-sheet, for each application so someone would be able to rebuild the server if I happened to get hit by a bus.

That is a lot of documentation to maintain.



There wasn’t enough justification to stand up automation around these applications, like Ansible, simply because I was only doing a few of them.

So, for a while, I wrote and tested code in environments that in no way mimicked their production deployment, and I spent a lot of time doing excessive documentation on doing those deployments.

But there is another way…

That other way is Docker.

What is Docker?

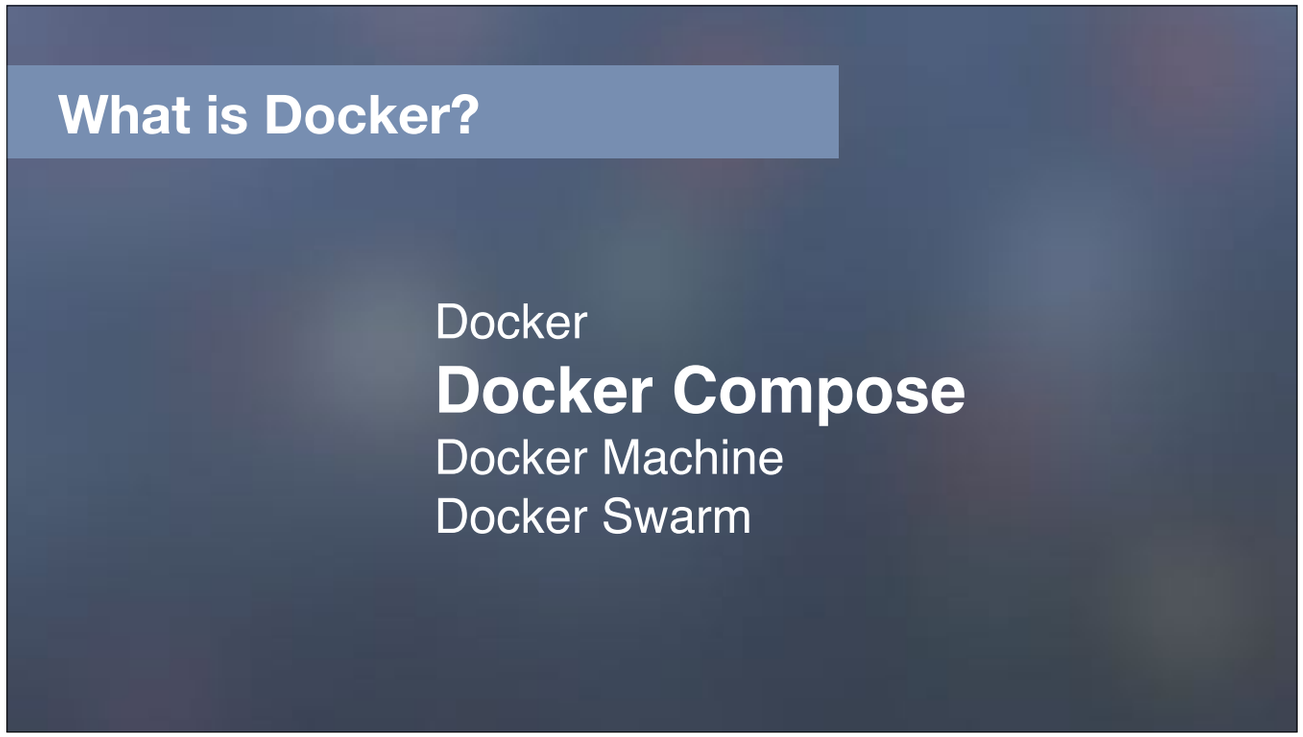
Docker
Docker Compose
Docker Machine
Docker Swarm

There are actually four components to the whole Docker solution.

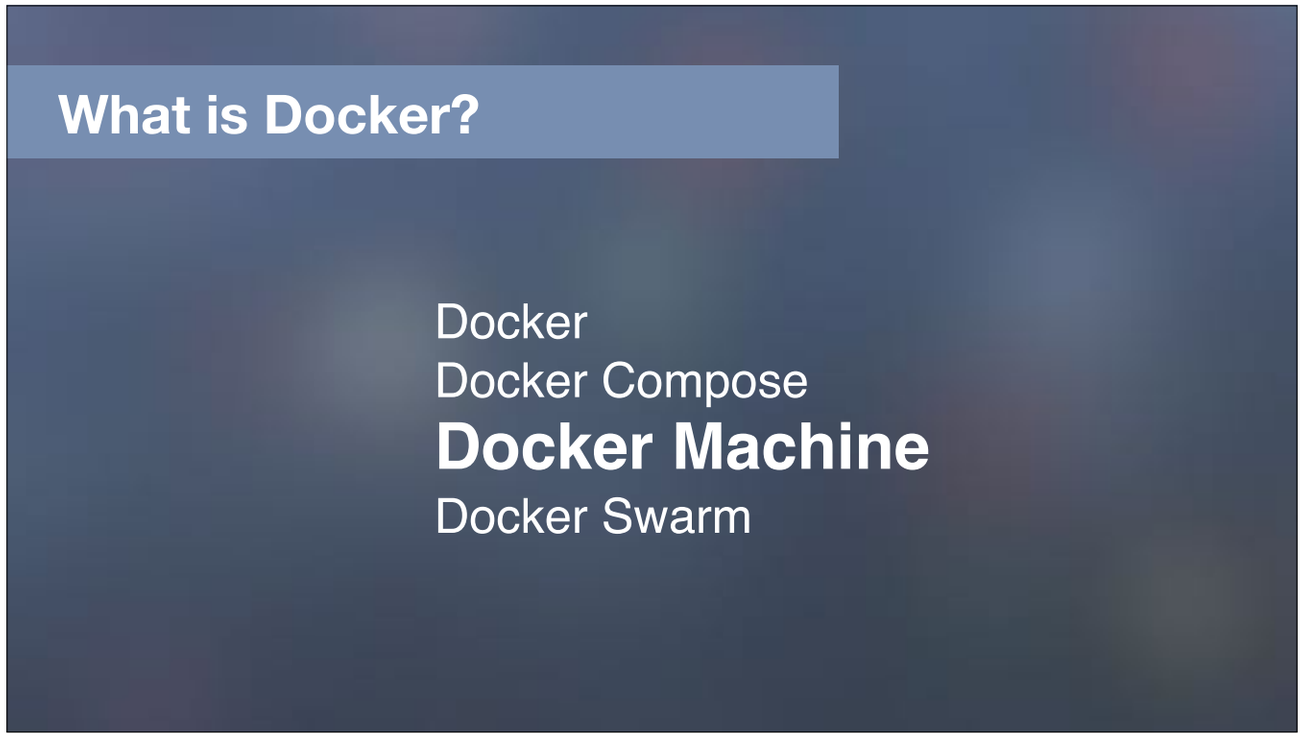
What is Docker?

Docker
Docker Compose
Docker Machine
Docker Swarm

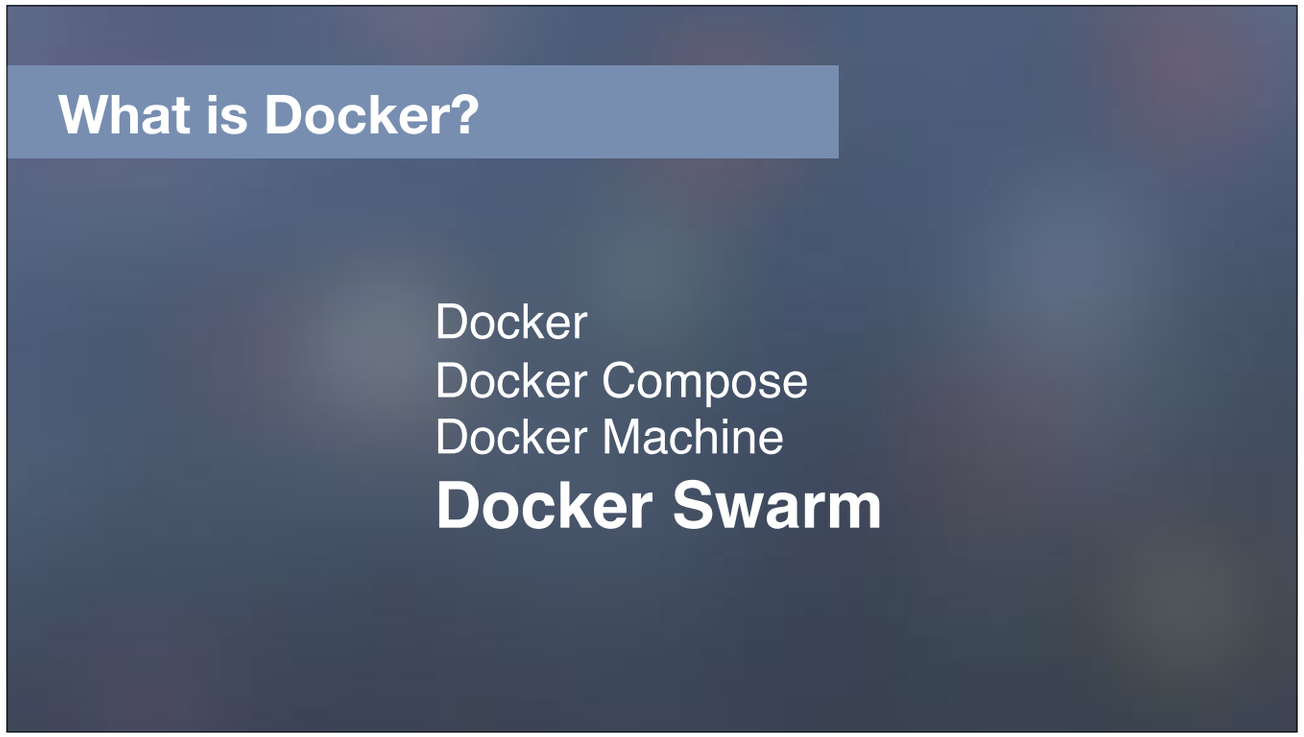
Docker is the utility that is all about building images and running containers. We'll get into that more in just a little bit.



Docker Compose, the namesake of this whole session, is an orchestration tool. Meaning that where Docker is what builds and runs images, Docker Compose defines those images, how they’re linked together, what other resources they’re attached to, etc.

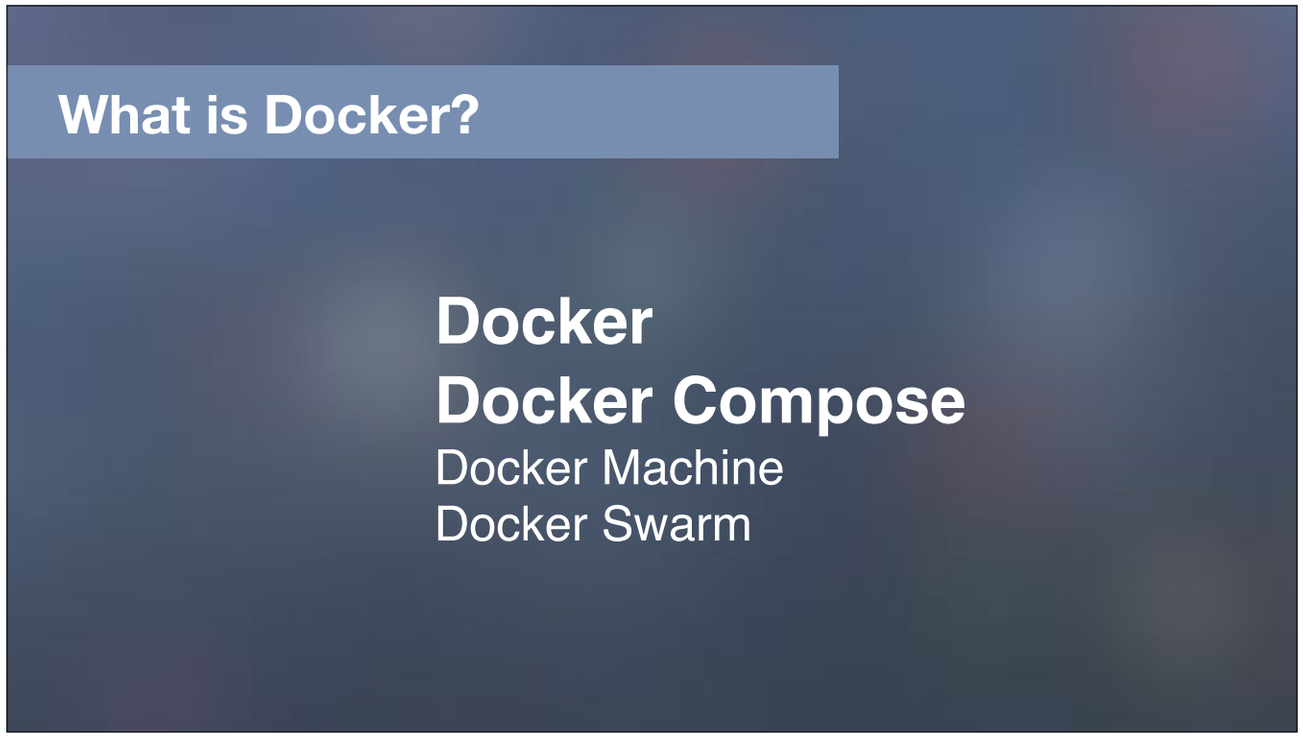


Now, the first two aren’t going to any good without having something to run your containers on. That’s called a Docker Host, and you can remotely create, provision, and manage Docker Hosts using Docker Machine. I use this tool for creating Docker hosts locally on my Mac right here, and for building and deploying on remote VMs either in the cloud or on my company network.



Docker Swarm is the next step above your single Docker Hosts. Swam allows you to have a bunch of hosts grouped together in a cluster. That means if you have multiple copies of a container running on a Swarm those instances will be spread out evenly across all those clustered hosts giving you greater performance.

That whole subject is way outside this session so don’t worry, we’re not covering it.

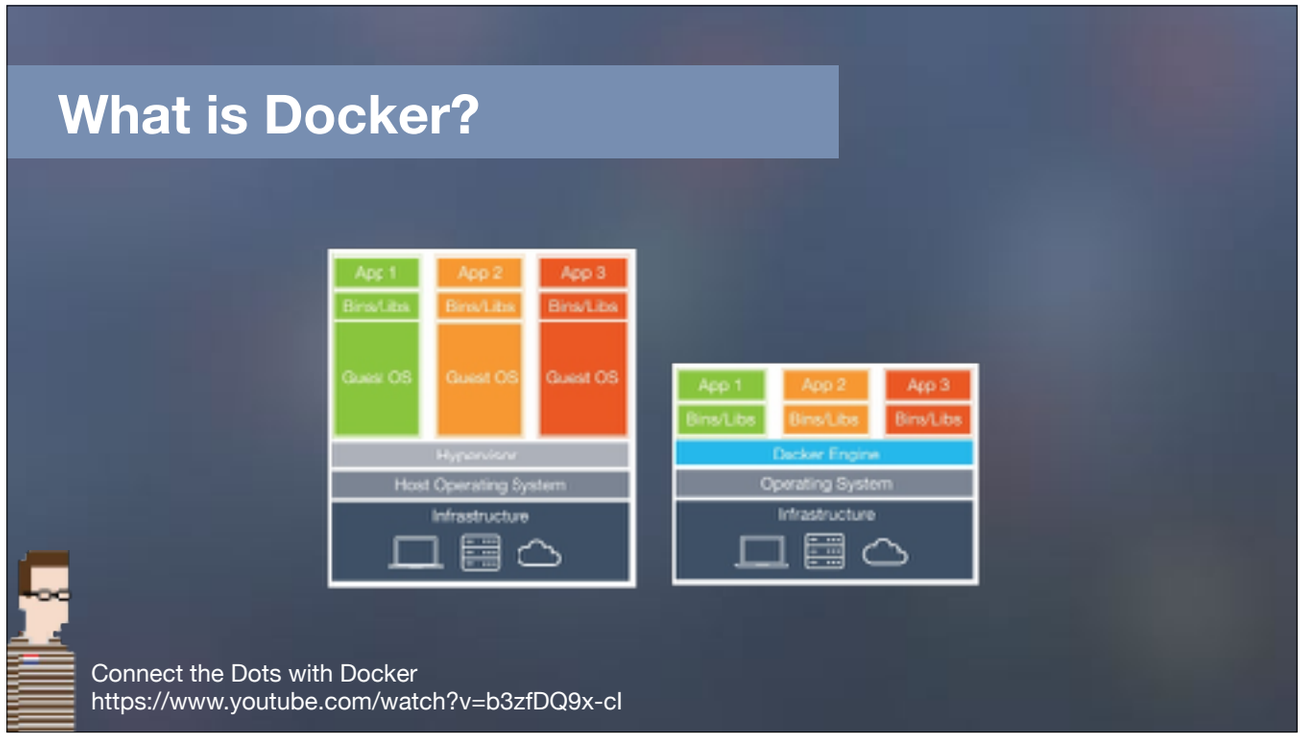


I’m only covering these two today, but when I mention a Docker Host you can safely assume I used Docker Machine. And while the presentation is about Docker Compose, we can’t do that without first getting into the core Docker utility.

What is Docker?

> … the ability to package and run an application
> in a loosely isolated environment called a container.

So, what is Docker?



I won’t go into great detail about Docker. There are a lot of resources out there if you want to learn more about this piece of the overall Docker solution, and I recommend you go back and watch Pepijn’s presentation from the 2015 conference, which you can find here.

There are a lot of cool things about Docker. If you look at the two diagrams here you can see that all the overhead of having the whole OS running is pretty much gone comparing a VM host to a Docker host, but each of the applications or services are still isolated from each other as if they were seperate systems.

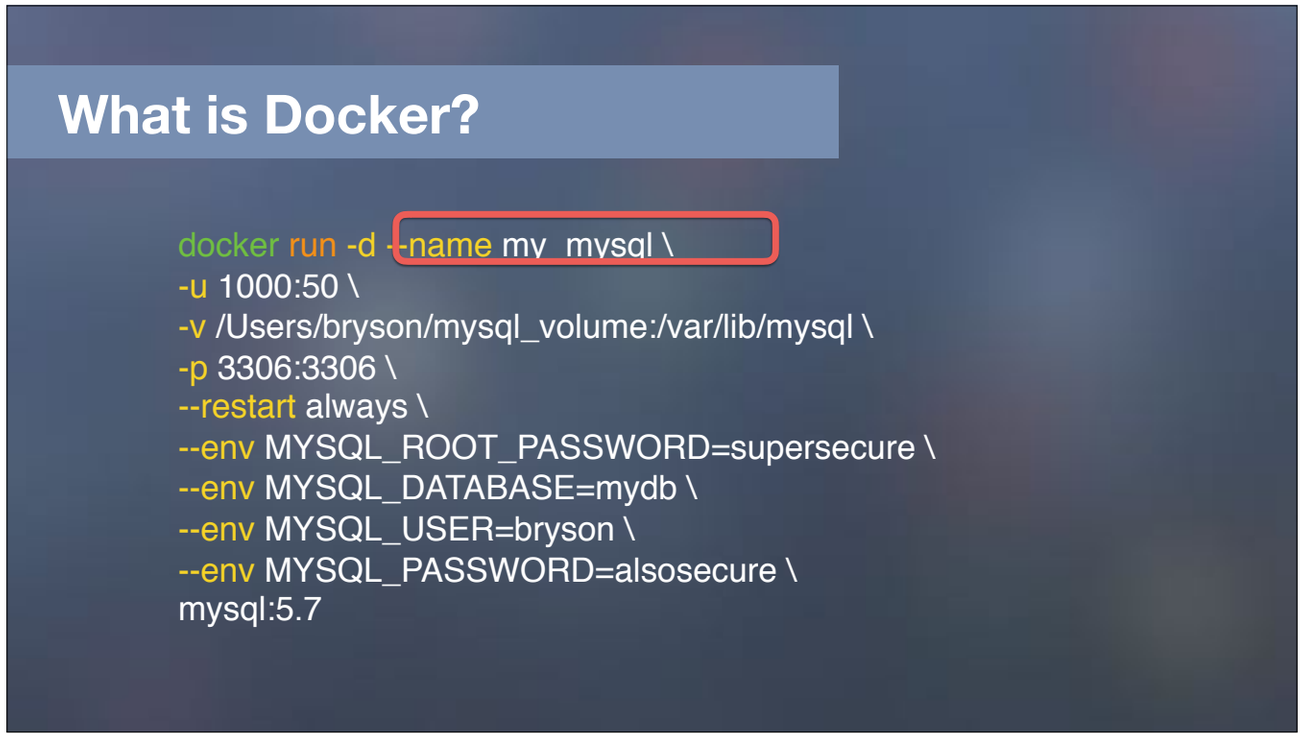
What is Docker?

```
docker run -d --name my_mysql \
-u 1000:50 \
-v /Users/bryson/mysql_volume:/var/lib/mysql \
-p 3306:3306 \
--restart always \
--env MYSQL_ROOT_PASSWORD=supersecure \
--env MYSQL_DATABASE=mydb \
--env MYSQL_USER=bryson \
--env MYSQL_PASSWORD=alsosecure \
mysql:5.7
```

Let’s assume that I created a Docker host on my Mac using an Ubuntu VM running in Virtualbox.

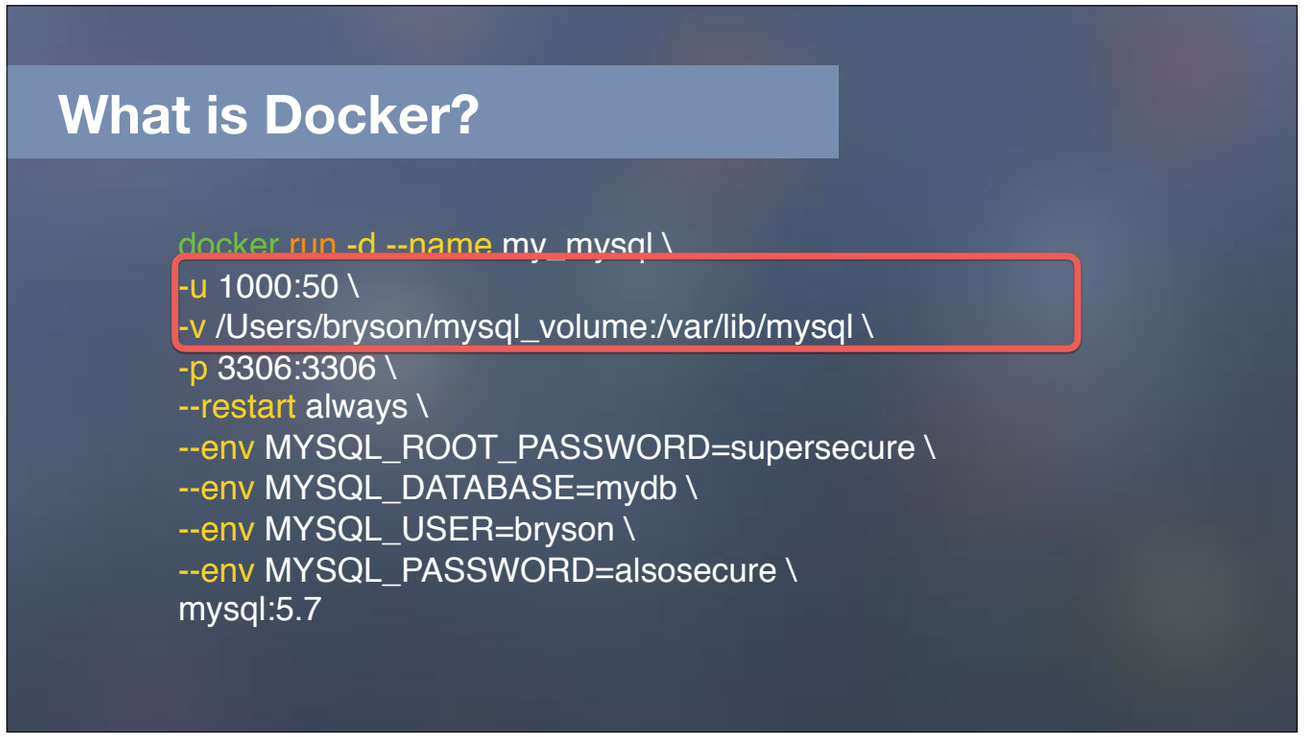
Using Docker, I can start up a container for a MySQL server to use with port 3306 exposed by running this command. Now if I were testing my web application in the debugger I could point it to an actual MySQL server on the localhost address and not a local SQL file.

There’s a lot going on there and I’ll cover the important points.



At the start of the command I'm using -d to say "run detached" which means it will launch the container and return me to the terminal. If I don't include -d it will be like I was running any other command and I'll see log output as it runs.

The —name argument gives my container a specific name instead of a randomly generated one.



When using Docker on the Mac with Virtualbox, it will mount your user directory as a shared volume. The -v flag lets me mount a directory on the Docker host that the container will have access to - and I'm mapping that directory to where MySQL writes all the data for the database. This allows my database to PERSIST whenever I tear the container down.

The -u flag is only there for a permissions fix because I'm saving data onto my Mac in my user folder that is being mounted through virtualbox. This isn't normally needed if you're not mounting a directory in the users folders.

What is Docker?

```
docker run -d --name my_mysql \
-u 1000:50 \
-v /Users/bryson/mysql_volume:/var/lib/mysql \
-p 3306:3306 \
--restart always \
--env MYSQL_ROOT_PASSWORD=supersecure \
--env MYSQL_DATABASE=mydb \
--env MYSQL_USER=bryson \
--env MYSQL_PASSWORD=alsosecure \
mysql:5.7
```

I want to be able to access MySQL from other systems. By default, container ports are only exposed externally so I need to map it to a port on the Docker host (which in this case is the same port).

What is Docker?

```
docker run -d --name my_mysql \
-u 1000:50 \
-v /Users/bryson/mysql_volume:/var/lib/mysql \
-p 3306:3306 \
--restart always \
--env MYSQL_ROOT_PASSWORD=supersecure \
--env MYSQL_DATABASE=mydb \
--env MYSQL_USER=bryson \
--env MYSQL_PASSWORD=alsosecure \
mysql:5.7
```

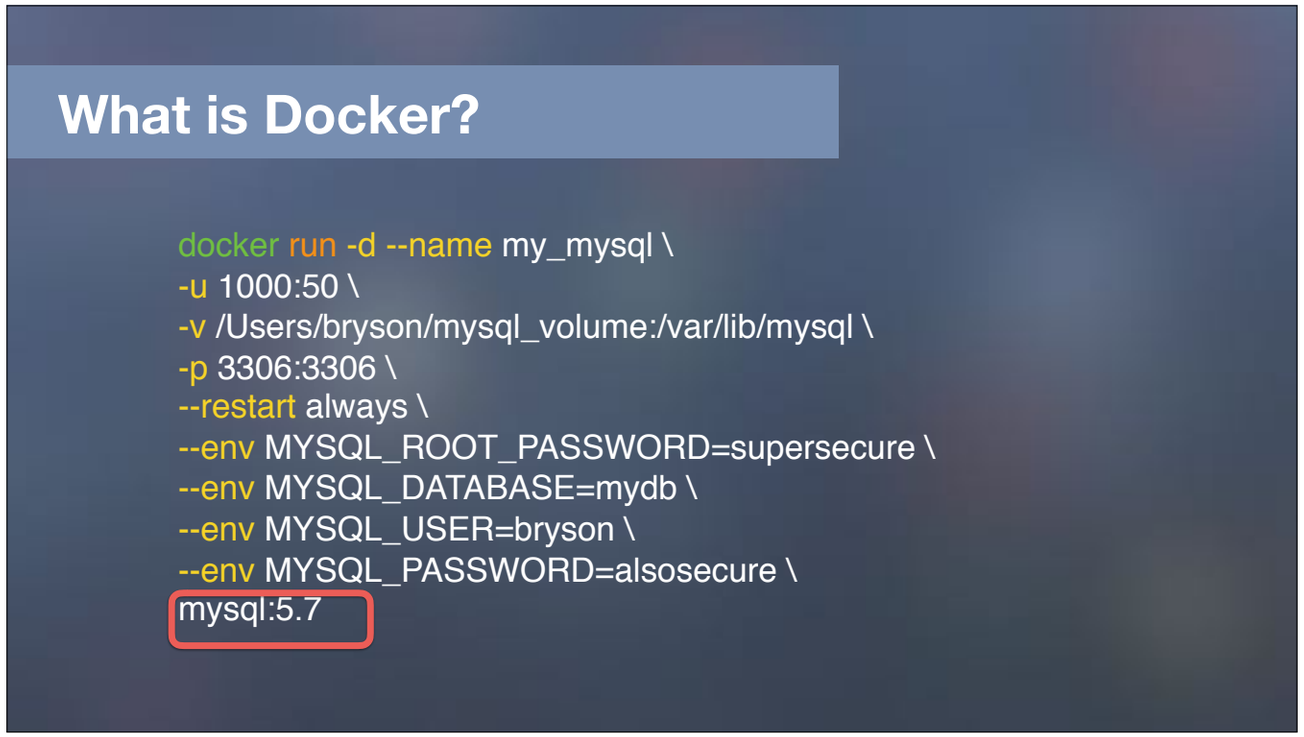
If the container crashes I want it to automatically restart always.

What is Docker?

```
docker run -d --name my_mysql \
-u 1000:50 \
-v /Users/bryson/mysql_volume:/var/lib/mysql \
-p 3306:3306 \
--restart always \
--env MYSQL_ROOT_PASSWORD=supersecure \
--env MYSQL_DATABASE=mydb \
--env MYSQL_USER=bryson \
--env MYSQL_PASSWORD=alsosecure \
mysql:5.7
```

These are the environment variables to configure MySQL. I'm giving it a root password, a database to create and the user that will be able to access that database.

Most Docker images have documented environment variables for you to set to configure them upon launch.

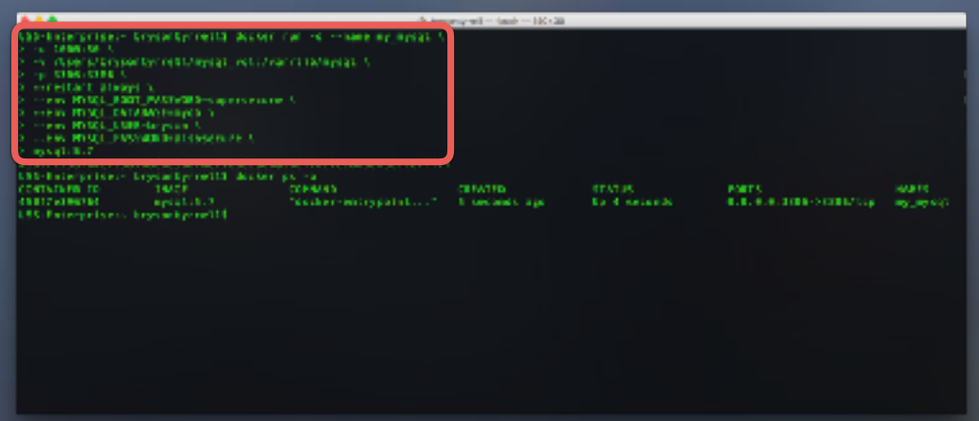


Lastly, this is the exact Docker image I want to run. I want it to run the MySQL image, and I want to specifically use the 5.7 version. This is going to automatically pull from Docker Hub where all the official images are hosted.

You've been listening to me say the words image and container in this whole thread and might be wondering why? What's the difference? Images are images - they aren't running. Containers are the running instances of a loaded image.

Think of it like a golden master image for the bygone days of Mac imaging. The GM isn't your managed endpoint, it's the basis that you load onto a Mac and then it becomes an active manged endpoint. Same concept.

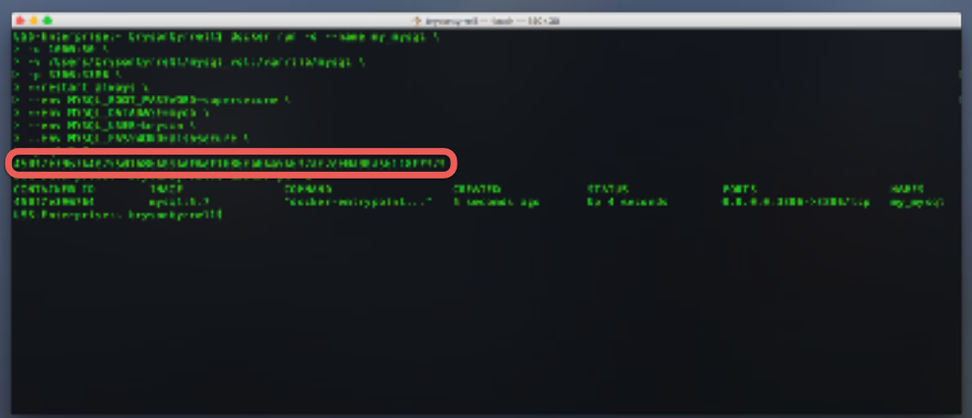
What is Docker?



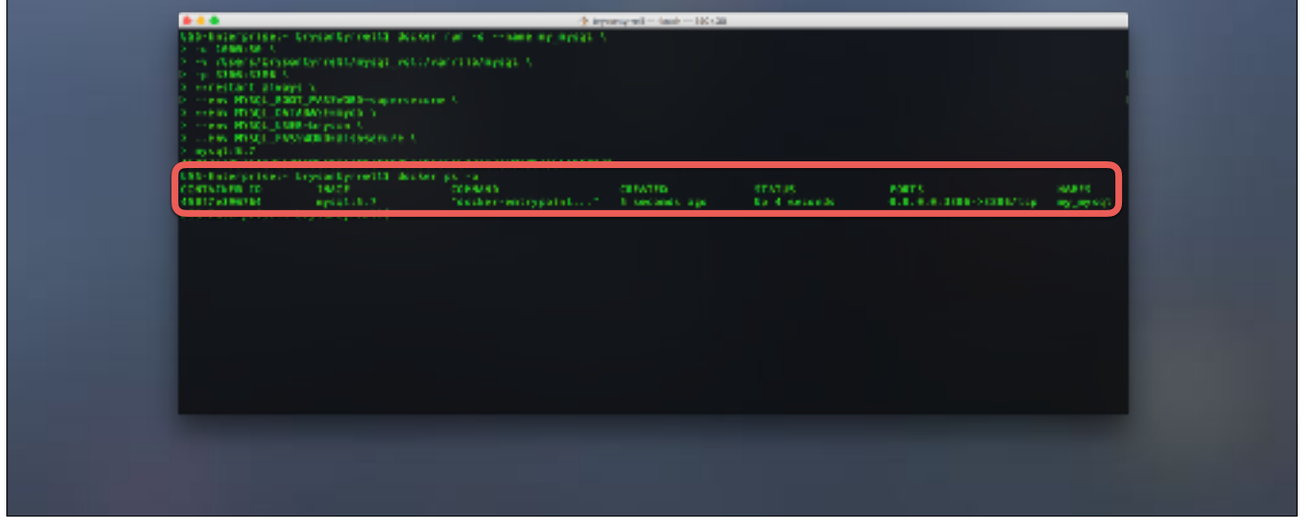
Here’s what that command looks like in my lovely green Terminal.

I’ve pasted it in here and executed it.

What is Docker?



When the container runs I get a generated ID. Everything in Docker gets an ID like this. You can interact with Docker resources by either this ID or the name that gets assigned to them.



Now I’m going to use ‘docker ps -a’ to show the running container on my Docker host and get some information about it.

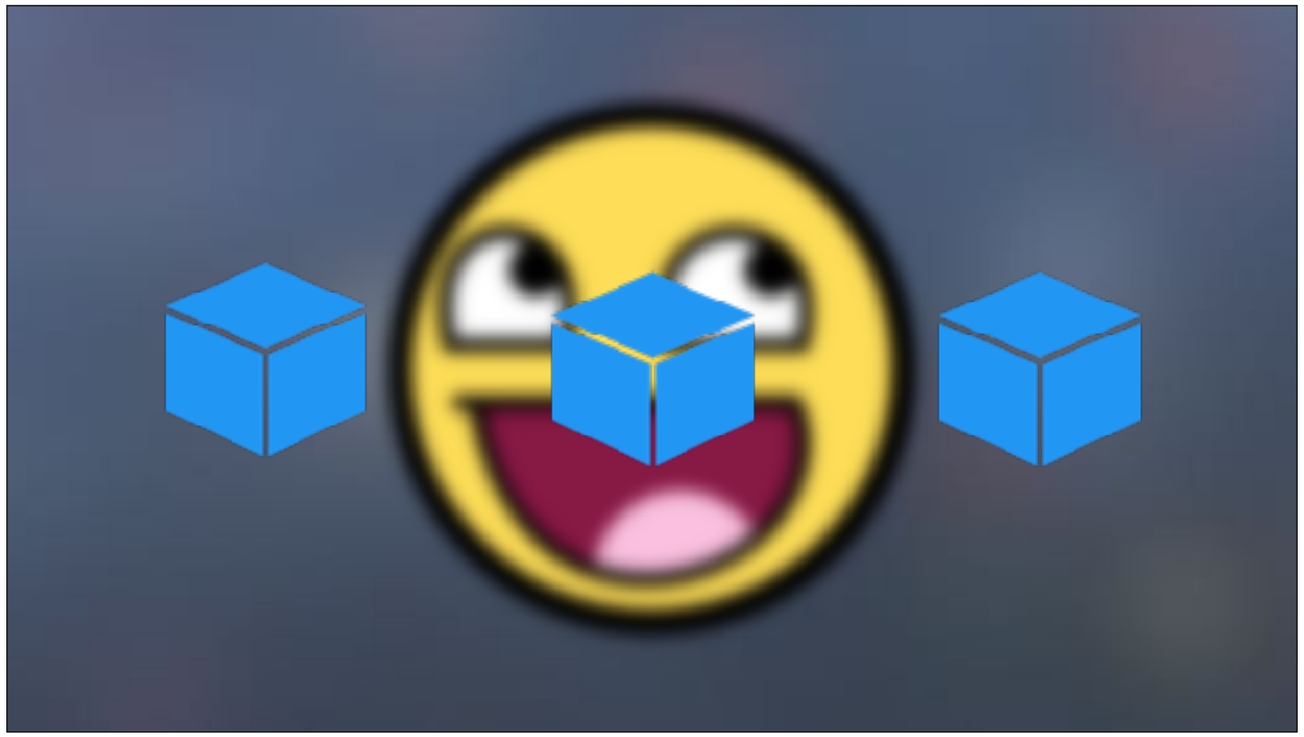
The ID in the table is actually a truncated version of that really long full ID from before. We got the image that it launched from, when it was created, how long its been up, and under ports we see that 3306 on the host is bound to 3306 on the container.

That was it. We now have a running MySQL server to develop against. And the sheer beauty of using Docker instead of a dedicated VM running MySQL or having MySQL installed on my Mac is that I can trash this and relaunch a new one clean with a couple keystrokes. Nothing to clean up.

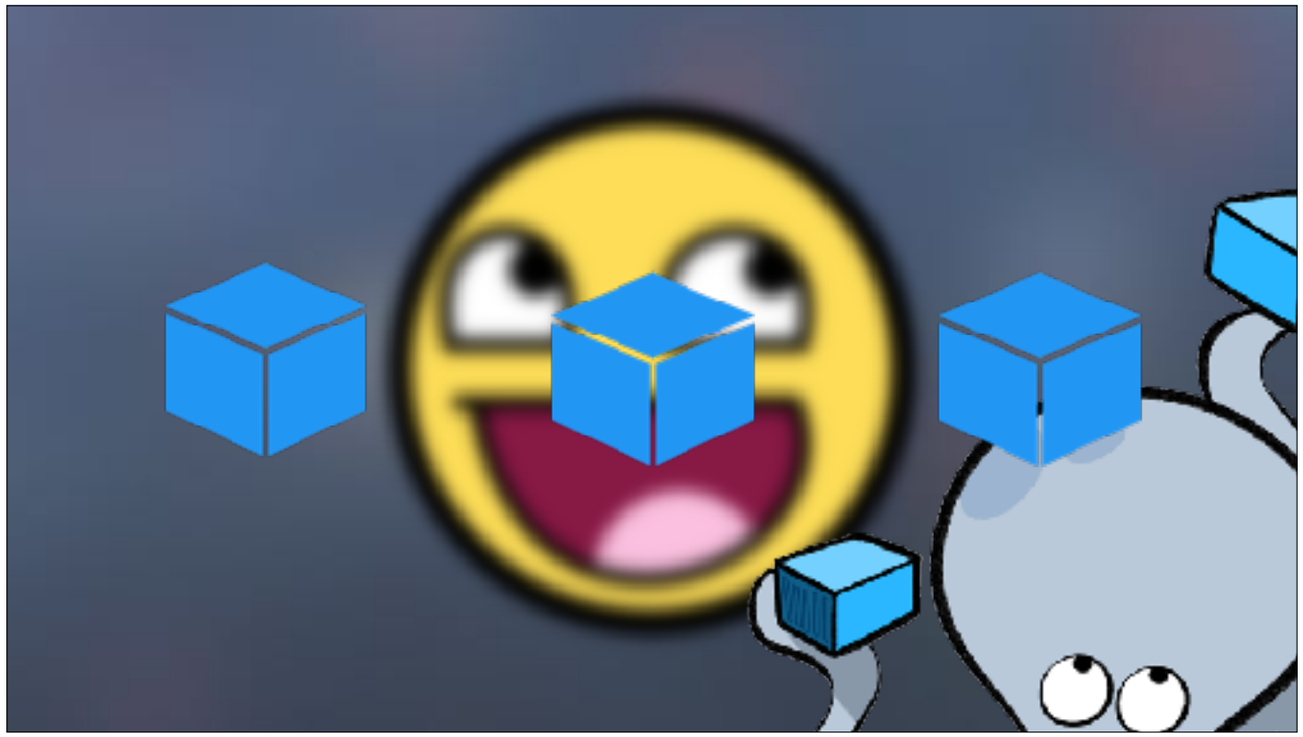
This is awesome. But…

That only covers MySQL. I need to do this with a web server as well, which there is an Nginx container available for. Still, different apps have different additional dependencies byond just the database and the web server.

My DNS registering service, its called QuickDNS, needs a bind server. I might also need a caching service like Redis so I can implement a queuing system.



Now we’re talking about a lot of different containers working together with my application. Really, I need to launch a full environment. Now, I could script all of this using the docker command like we saw before and have that bring everything up in the right order…



…but there is a better way.

What is Docker Compose?

> …is a tool for defining and running multi-container Docker applications.

We’re finally talking about Docker Compose.

Great, but how does it do that?

What is Docker Compose?

> …you use a Compose file to configure your application's services. Then, using a single command, you create and start all the services from your configuration.

A compose file and a command? What does that look like?

What is Docker Compose?

docker-compose.yml

A Docker Compose file is a Yaml document. What is Yaml? It's hipster JSON. There is no excuse for it. Someone decided to take a perfectly defined data format and make it jazzy.

Still, Yaml is actually a fully compatible superset of JSON. While there are many differences, the key one is that Yaml uses whitespace indentation for formatting instead of curly braces.

A default Compose file is going to be named docker-compose.yml. You're going to see examples in just a moment.

With our compose file, we only need to run one command to have it fire up everything that is described within.

What is Docker Compose?

```
docker run -d --name my_mysql \
-u 1000:50 \
-v /Users/bryson/mysql_volume:/var/lib/mysql \
-p 3306:3306 \
--restart always \
--env MYSQL_ROOT_PASSWORD=supersecure \
--env MYSQL_DATABASE=mydb \
--env MYSQL_USER=bryson \
--env MYSQL_PASSWORD=alsosecure \
mysql:5.7
```

Here’s our docker command from earlier.

Translated to a Compose file, it looks like…

What is Docker Compose?

```
version: '2'
services:
  my_mysql:
    user: '1000:50'
    image: 'mysql:5.7'
    volumes:
      - '/Users/brysontyrrell/Development/docker/mysql/mysql_volume:/var/lib/mysql'
    restart: 'always'
    ports:
     - '3306:3306'
    environment:
      MYSQL_ROOT_PASSWORD: ‘supersecure'
      MYSQL_DATABASE: 'mydb'
      MYSQL_USER: ‘bryson'
      MYSQL_PASSWORD: ‘alsosecure'
```

Not only does that look cleaner, but it also looks so much easier to maintain, and we’re still only talking about doing this with a single service.

Now all of a sudden, there’s this feeling…


www.jamfsoftware.com

What is Docker Compose?

```
version: '2'
services:
  my_mysql:
    user: '1000:50'
    image: 'mysql:5.7'
    volumes:
      - '/Users/brysontyrrell/Development/docker/mysql/mysql_volume:/var/lib/mysql'
    restart: 'always'
    ports:
     - '3306:3306'
    environment:
      MYSQL_ROOT_PASSWORD: ‘supersecure'
      MYSQL_DATABASE: 'mydb'
      MYSQL_USER: ‘bryson'
      MYSQL_PASSWORD: ‘alsosecure'
```

The version is required on every Compose file. This refers to the compose file specifically, not the docker-compose binary. There are three versions, and with each version there has been support for additional functionality for how Docker Compose builds images and launches containers.

And this version also ties to the version of the docker-compose binary you’re running. Older versions won’t support version 3 compose file options, but newer versions of the binary WILL support older compose file versions - so you don’t have to use the latest version if you don’t want to, but you might miss out on cool new features.

What is Docker Compose?

```
version: '2'
services:
  my_mysql:
    user: '1000:50'
    image: 'mysql:5.7'
    volumes:
      - '/Users/brysontyrrell/Development/docker/mysql/mysql_volume:/var/lib/mysql'
    restart: 'always'
    ports:
      - '3306:3306'
    environment:
      MYSQL_ROOT_PASSWORD: ‘supersecure'
      MYSQL_DATABASE: 'mydb'
      MYSQL_USER: ‘bryson'
      MYSQL_PASSWORD: ‘alsosecure'
```

This next section describes all the services that are going to be built and launched. Note that I said services. Right now we only have our MySQL container described, but we can have as many as we want within one compose file and it will build and launch them all.

What is Docker Compose?

```
version: '2'
services:
  my_mysql:
    user: '1000:50'
    image: 'mysql:5.7'
    volumes:
      - '/Users/brysontyrrell/Development/docker/mysql/mysql_volume:/var/lib/mysql'
    restart: 'always'
    ports:
      - '3306:3306'
    environment:
      MYSQL_ROOT_PASSWORD: ‘supersecure'
      MYSQL_DATABASE: 'mydb'
      MYSQL_USER: ‘bryson'
      MYSQL_PASSWORD: ‘alsosecure'
```

Moving in one level, we have the full description of our MySQL container. The key for each service becomes the the name of the container.

What is Docker Compose?

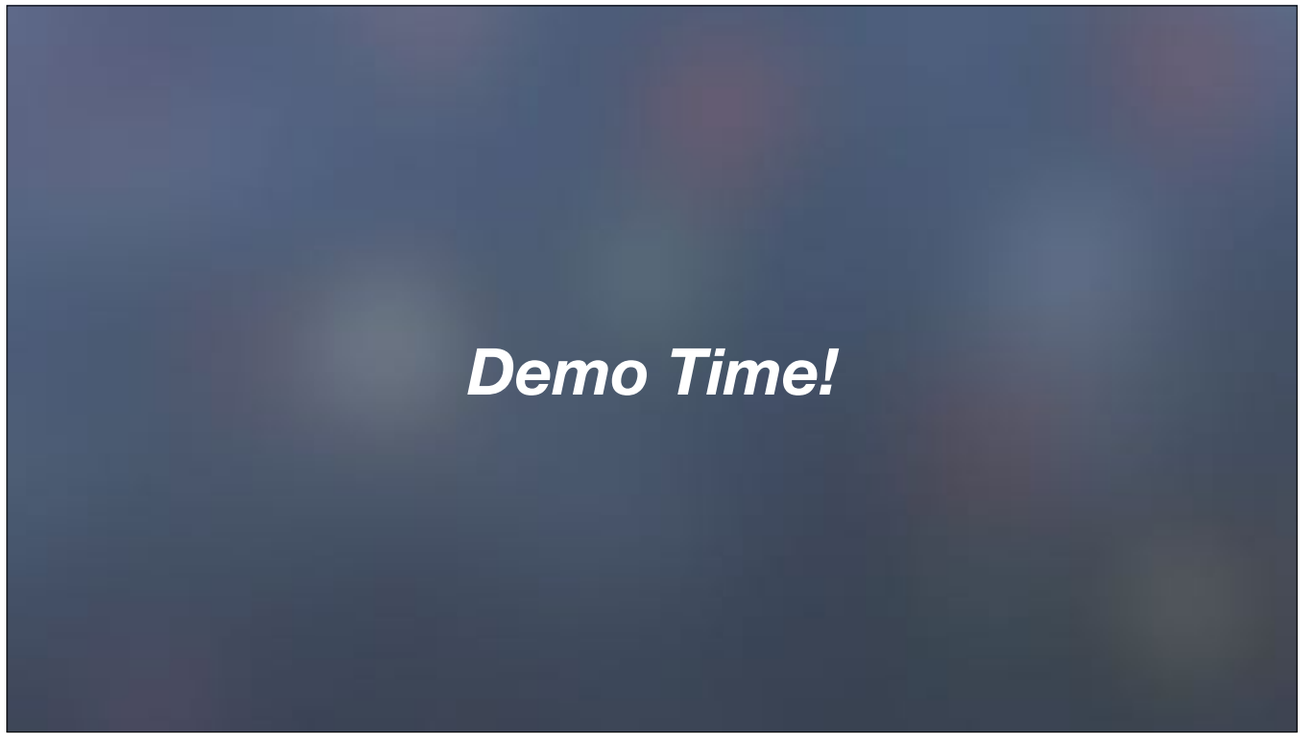
```
version: '2'
services:
  my_mysql:
    user: '1000:50'
    image: 'mysql:5.7'
    volumes:
      - '/Users/brysontyrrell/Development/docker/mysql/mysql_volume:/var/lib/mysql'
    restart: 'always'
    ports:
      - '3306:3306'
    environment:
      MYSQL_ROOT_PASSWORD: ‘supersecure'
      MYSQL_DATABASE: 'mydb'
      MYSQL_USER: ‘bryson'
      MYSQL_PASSWORD: ‘alsosecure'
```

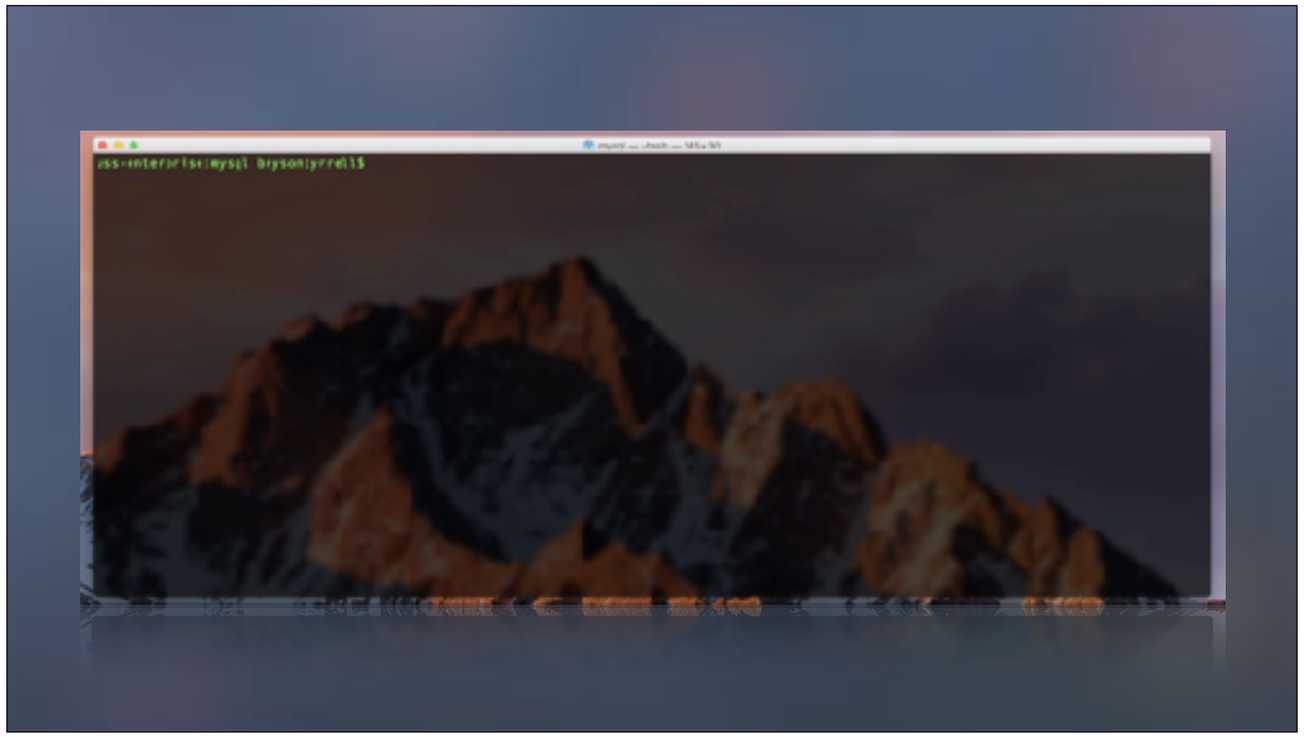
All of the arguments from the docker command are nested under the service

What is Docker Compose?

```
version: '2'
services:
  my_mysql:
    user: '1000:50'
    image: 'mysql:5.7'
    volumes:
      - '/Users/brysontyrrell/Development/docker/mysql/mysql_volume:/var/lib/mysql'
    restart: 'always'
    ports:
     - '3306:3306'
    environment:
      MYSQL_ROOT_PASSWORD: ‘supersecure'
      MYSQL_DATABASE: 'mydb'
      MYSQL_USER: ‘bryson'
      MYSQL_PASSWORD: ‘alsosecure'
```

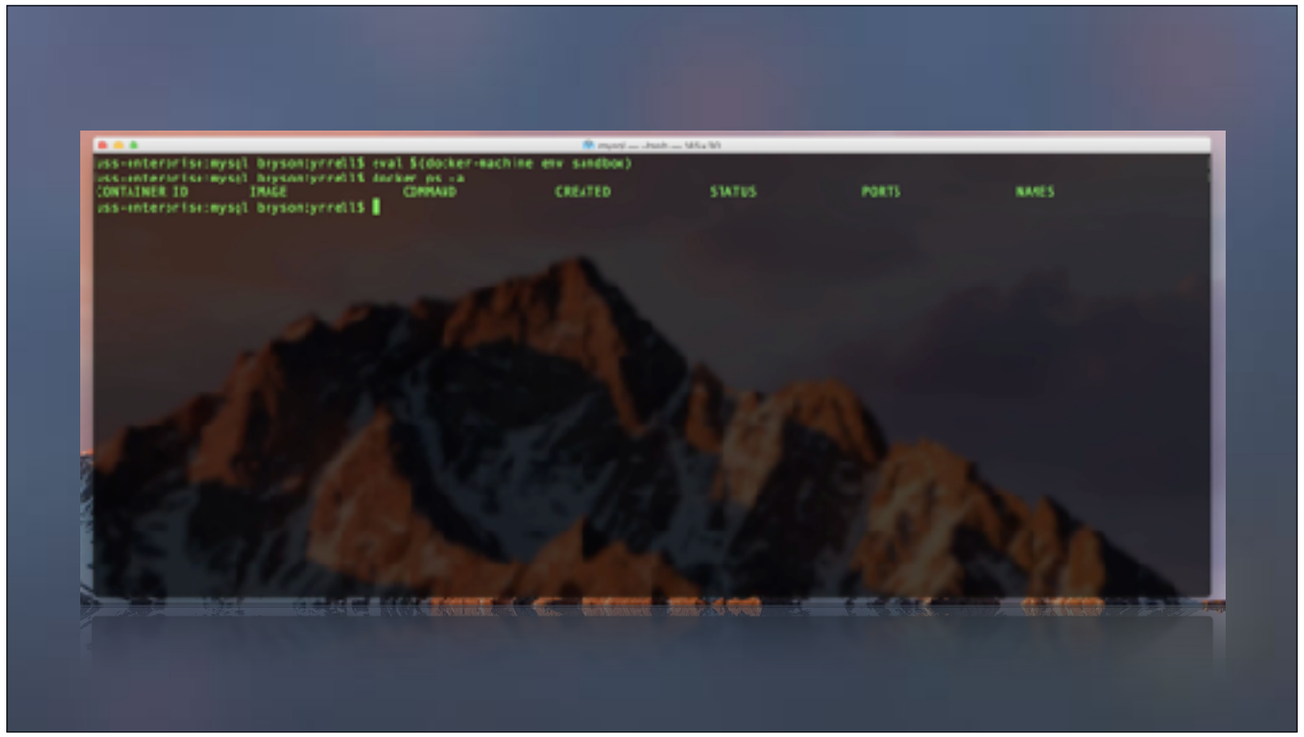
Here are all our environment variables, but in a minute I’ll show you an alternative to having them in the compose file.


Demo Time!



I’m going to use all three Docker command line tools together - Machine, Compose, Docker.

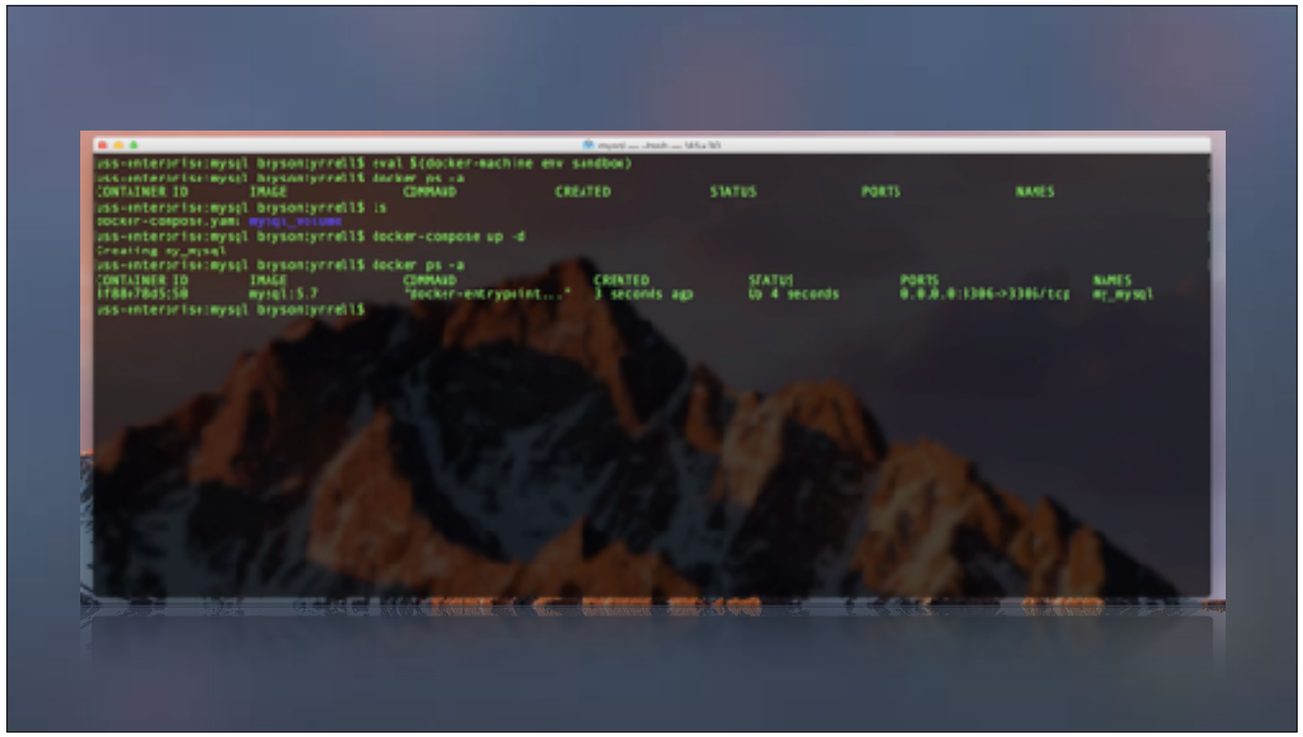
I have a Docker host called “sandbox” that’s running in VirtualBox on my Mac. I’m going to connect to it using ‘docker-machine’ and use ‘docker ps’ to see if there are any running containers.



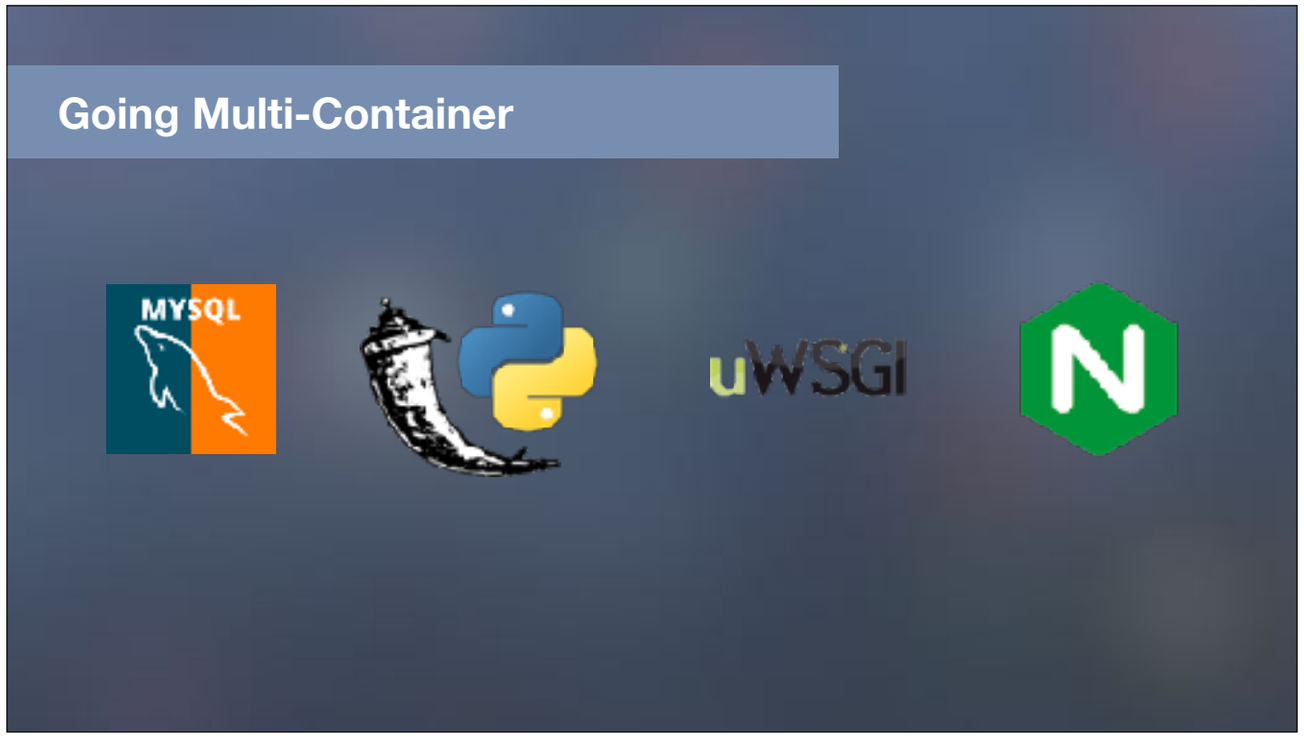
In my current directory I have the compose file we were just looking at as well as the local directory I'm using for the mounted volume that MySQL will store all the database information into.

I'm now going to launch the MySQL container by running 'docker-compose up' and view it by running 'docker ps' again.

We can see that the port mapping from the container to the host is in place just like before.

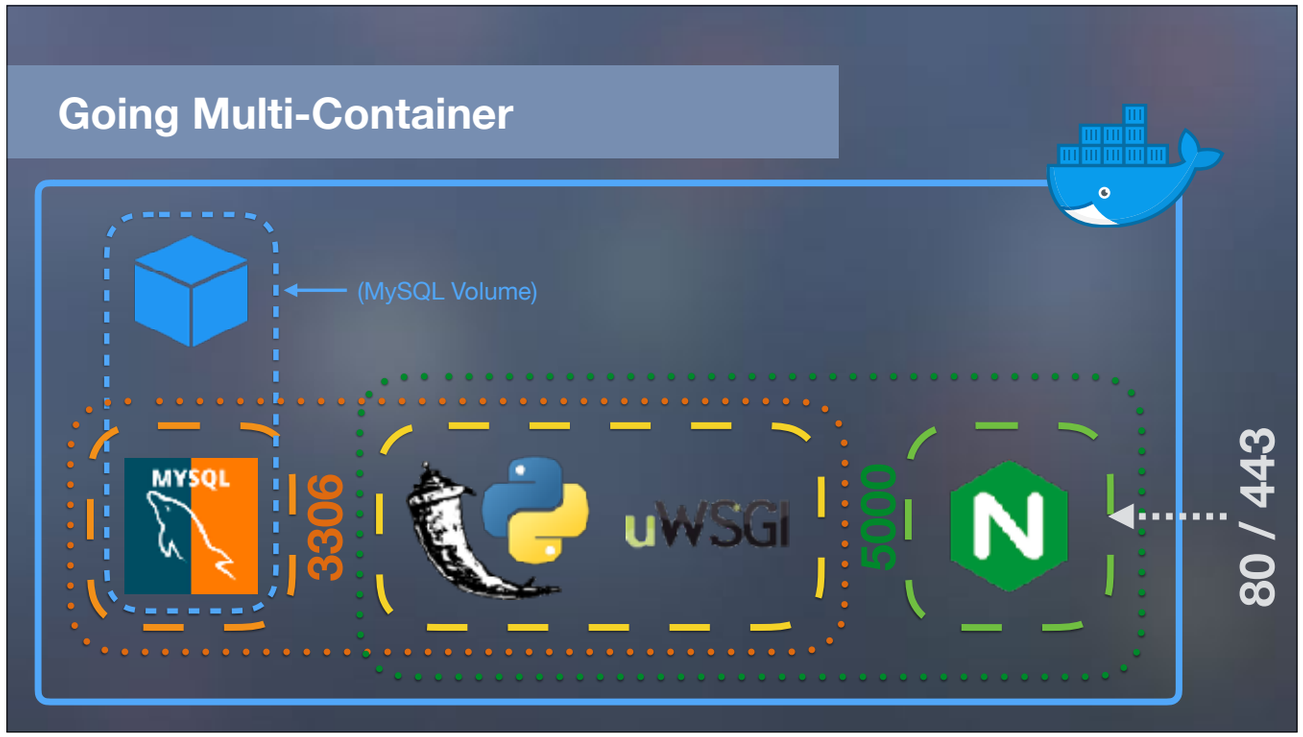


Now that the container is running I can continue to use ‘docker’ to inspect the individual running containers. Right now I’m getting the logs so I can see everything that executed on startup.

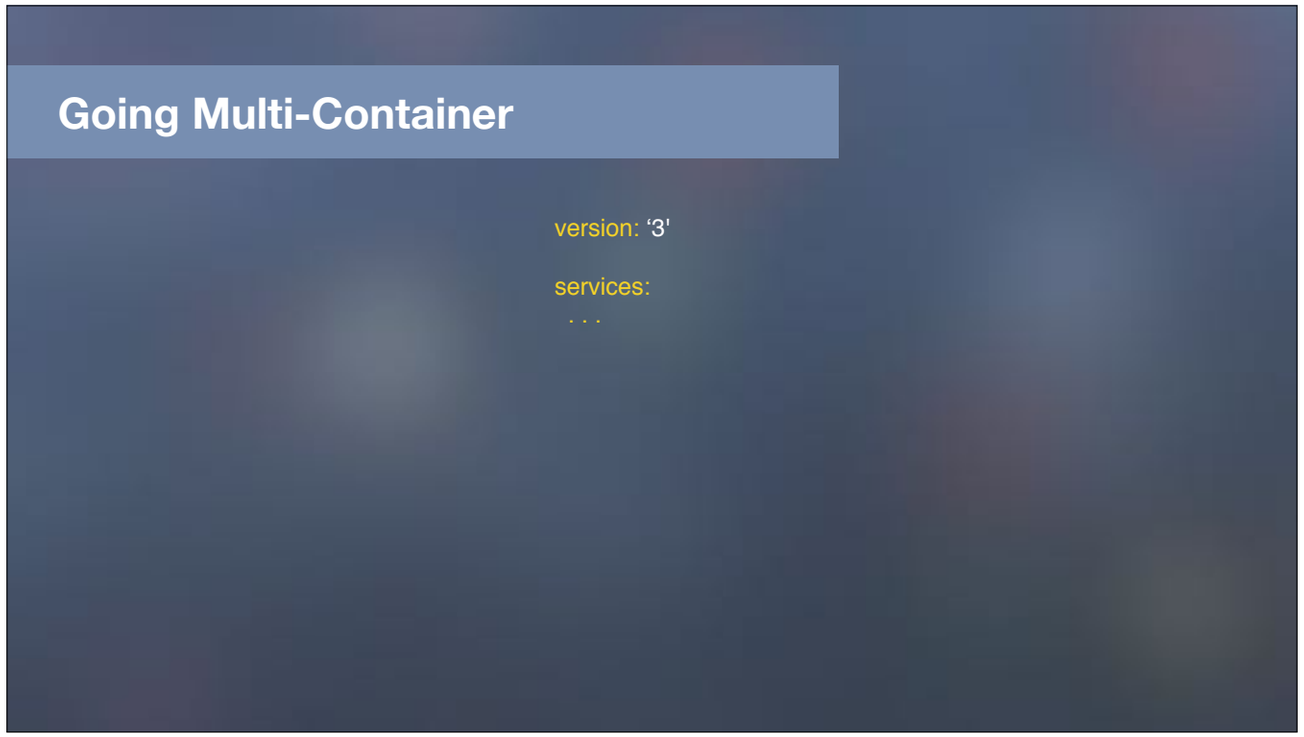


We covered one service in the example just now, but let’s scale that to a Docker Compose setup that will cover all the elements of one of those web applications I described earlier.

And let’s also set this up so that only the pieces that need to talk to each other can, and we’re persisting the database in a volume that isn’t on our Mac. Those elements are Docker networks and Docker volumes.



1) Let’s break it down visually. First, we have our Docker host.
2) Then we have the containers. There are three: MySQL, the web application, and the web server.
3) Now we’re going to create two networks that will restrict each container’s ability to interact with the other. MySQL and Nginx can only talk to the web app - they can’t talk to each other. Because we’re only mapping the HTTP ports from the Nginx container to the host, absolutely no external traffic will ever be able to touch the web app or MySQL.
4) The volume is going to follow the same principal. It will be only linked to the MySQL container. The web app and Nginx will not be able to access any of the database files.



We’re going to update that original Compose file to achieve this configuration. We’re going to be adding additional sections before adding the two new containers for the application and Nginx.

Going Multi-Container: Networks

```
version: ‘3'

services:
  . . .

networks:
  proxy:
  db:
```

In the diagram we had two networks defined to isolate our MySQL and web containers from each other.

We’re going to label them ‘proxy’ and ‘db’ respectively.

Going Multi-Container: Volumes

```
version: ‘3'

services:
  . . .

networks:
  proxy:
  db:

volumes:
  mysql:
```

Instead of mounting a location on my Mac, or even directly on the Docker host, we will create a data volume for the MySQL container to use.

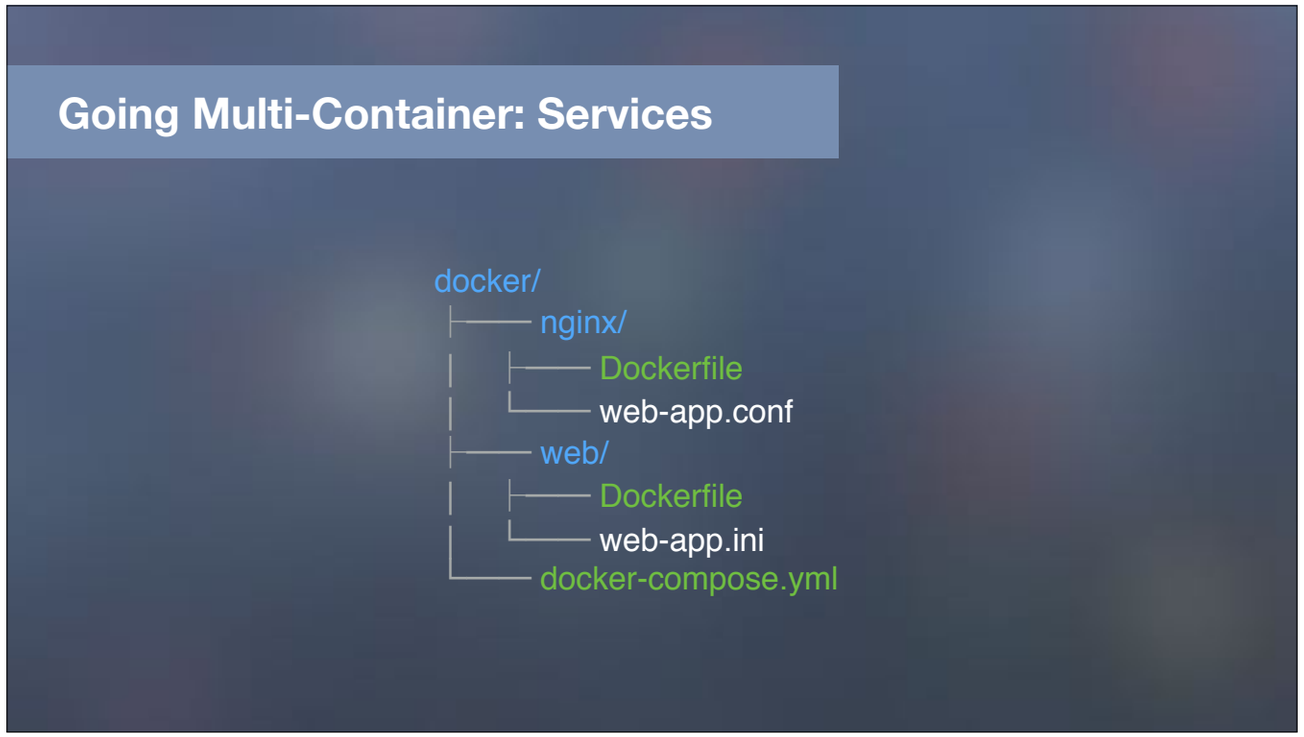
Going Multi-Container: Volumes

```
version: '3'

services:
  . . .
  mysql:
    volumes:
      - mysql:/var/lib/mysql

networks:
  proxy:
  db:

volumes:
  mysql:
```

There's only a small change needed for the MySQL container's volumes key to reference the new volume.

If you find it confusing to share names across items know you don't have to do what I do. You can name them whatever you want.

Going Multi-Container: Services



As mentioned before, Docker Compose builds images in addition to launching containers. When you need to customize an image beyond what it provided from what is served from Docker Hub you're going to use a Dockerfile to handle that.

Again, I won't get into Dockerfiles very deep, but I'm going to show you the two included here and what they're doing in just a bit.

Going Multi-Container: Services

```
services:
  . . .

  nginx:
    build: ./nginx
    restart: always
    ports:
      - "80:80"
      - “443:443"
    networks:
      - proxy
    depends_on:
      - web

. . .
```

This is the section that details the Nginx container.

Instead of just an ‘image’ key we’re using a ‘build’ key and telling it which relative directory contains the Dockerfile to use.

Going Multi-Container: Services

```
services:
  . . .

  nginx:
    build: ./nginx
    restart: always
    ports:
      - "80:80"
      - “443:443"
    networks:
      - proxy
    depends_on:
      - web

. . .
```

We’ve mapped ports 80 and 443 from the Nginx container to ports 80 and 443 on the Docker host. These will be the only exposed ports, and they will only be on the web server container, so no other communication will be allowed in.

Going Multi-Container: Services

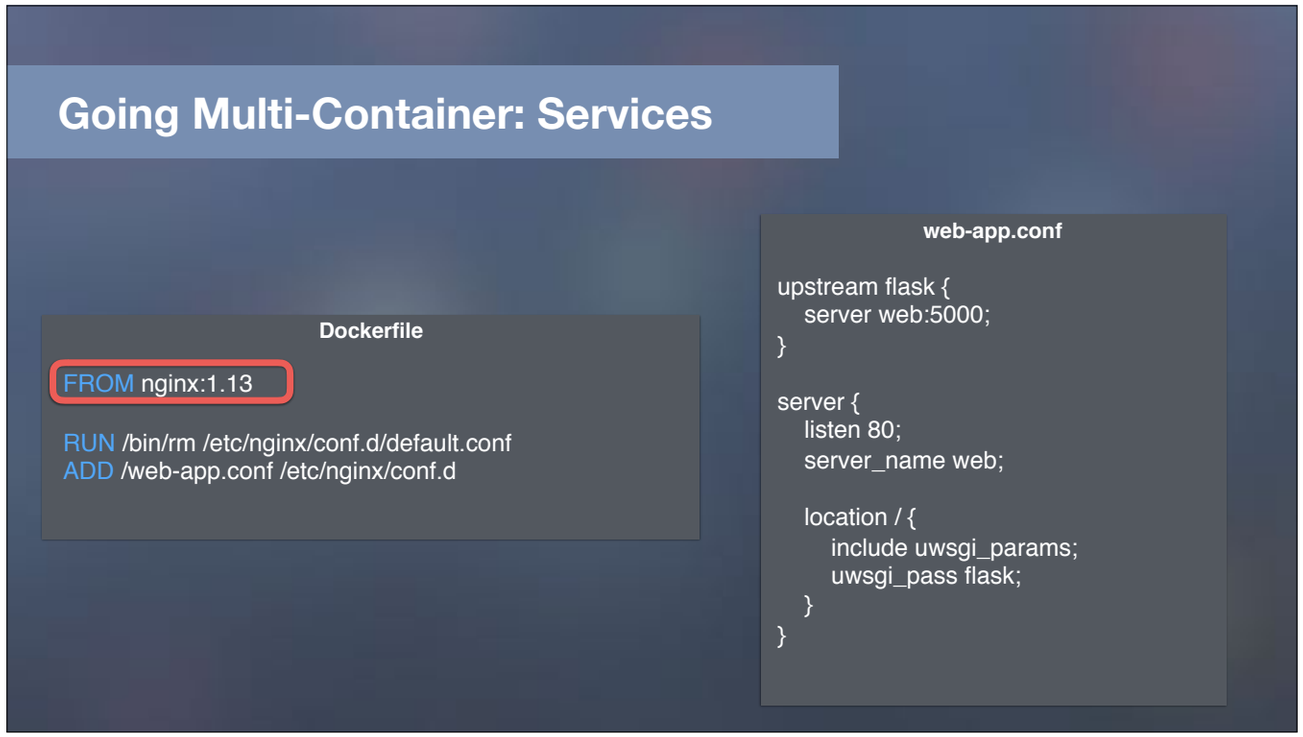
```
services:
  . . .

  nginx:
    build: ./nginx
    restart: always
    ports:
      - "80:80"
      - “443:443"
    networks:
      - proxy
    depends_on:
      - web

. . .
```

The Nginx container is attached to our ‘proxy’ network so the only other container it will be able to communicate with is the ‘web’ container once we configure that.

Going Multi-Container: Services

```
services:
  . . .

  nginx:
    build: ./nginx
    restart: always
    ports:
      - "80:80"
      - “443:443"
    networks:
      - proxy
    depends_on:
      - web

. . .
```

This key tells the container what other containers must be started before it starts. This prevents the web server from being available before the backend application is ready.



For Nginx the only customization is the configuration file that forwards port 80 traffic to port 5000 of the web app container.

Remember how we named the web app container ‘web’? That also becomes its alias on the Docker host network. We can direct traffic into the container just by passing the container’s name instead of a domain name or IP address.

In our original Compose file the ‘image’ tag pointed to the official Docker Hub MySQL image. The FROM statment does the same.

Going Multi-Container: Services

Dockerfile

```
FROM nginx:1.13

RUN /bin/rm /etc/nginx/conf.d/default.conf
ADD /web-app.conf /etc/nginx/conf.d
```

web-app.conf

```
upstream flask {
    server web:5000;
}

server {
    listen 80;
    server_name web;

    location / {
        include uwsgi_params;
        uwsgi_pass flask;
    }
}
```

The RUN statment executes a shell command. This is removing the default Nginx configuration.

Going Multi-Container: Services

Dockerfile

```
FROM nginx:1.13

RUN /bin/rm /etc/nginx/conf.d/default.conf
ADD /web-app.conf /etc/nginx/conf.d
```

web-app.conf

```
upstream flask {
    server web:5000;
}

server {
    listen 80;
    server_name web;

    location / {
        include uwsgi_params;
        uwsgi_pass flask;
    }
}
```

The ADD statment will copy content into the image from an external source. In this case, it’s the web-app.conf file on the right side that’s going to be added.

Going Multi-Container: Services

```
version: '3'

services:
  . . .
  web:
    build: ./web
    restart: always
    networks:
      - proxy
      - db
    depends_on:
      - mysql

. . .
```

The web container is very similar to the Nginx container.

It is a part of ‘proxy’ network to be able to communicate to the Nginx container over port 5000, and it is a part of the ‘db' network to communicate with MySQL over port 3306.

Going Multi-Container: Services

```
version: '3'

services:
  . . .
  web:
    build: ./web
    restart: always
    networks:
      - proxy
      - db
    depends_on:
      - mysql

. . .
```

The web container depends on the ‘mysql’ container being started before it does. The ‘nginx’ containet depends on the ‘web’ container. We’ve built a dependency chain into the startup process which ensures everything comes up in the correct order.

Going Multi-Container: Services

Dockerfile

```
FROM ubuntu:16.04

RUN /usr/bin/apt-get update -q && \
    /usr/bin/apt-get install -qqy build-essential git && \
    /usr/bin/apt-get install -qqy python-pip python-dev && \
    /usr/bin/apt-get install -qqy uwsgi uwsgi-plugin-python && \
    /usr/bin/apt-get clean && \
    /bin/rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

ADD /web-app.ini /etc/uwsgi/apps-enabled/

RUN /bin/mkdir /opt/web-app && \
    cd /opt/web-app && \
    /usr/bin/git clone https://github.com/application.git /opt/web-app && \
    /usr/bin/pip install -r requirements.txt && \
    /bin/chown -R www-data:www-data /opt/web-app

CMD ["uwsgi", "--ini", "/etc/uwsgi/apps-enabled/web-app.ini"]
```

This is a much more complex Dockerfile. In it, we are installing all of the system requirements for the application, copying the uWSGI configuration file, and then installing the application itself from a GitHub repository.

The base image I’m using this time is the official Ubuntu 16.04 image.

Going Multi-Container: Services

Dockerfile

```
FROM ubuntu:16.04

RUN /usr/bin/apt-get update -q && \
    /usr/bin/apt-get install -qqy build-essential git && \
    /usr/bin/apt-get install -qqy python-pip python-dev && \
    /usr/bin/apt-get install -qqy uwsgi uwsgi-plugin-python && \
    /usr/bin/apt-get clean && \
    /bin/rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

ADD /web-app.ini /etc/uwsgi/apps-enabled/

RUN /bin/mkdir /opt/web-app && \
    cd /opt/web-app && \
    /usr/bin/git clone https://github.com/application.git /opt/web-app && \
    /usr/bin/pip install -r requirements.txt && \
    /bin/chown -R www-data:www-data /opt/web-app

CMD ["uwsgi", "--ini", "/etc/uwsgi/apps-enabled/web-app.ini"]
```

The first RUN command uses apt-get to install all system level packages and dependencies.

Every statement in a Dockerfile adds a layer onto it. The more layers, with the more changes between layers, the larger the resulting Docker image. By chaining together related system commands you can reduce the total size of the layer within the image.

Going Multi-Container: Services

Dockerfile

```
FROM ubuntu:16.04

RUN /usr/bin/apt-get update -q && \
    /usr/bin/apt-get install -qqy build-essential git && \
    /usr/bin/apt-get install -qqy python-pip python-dev && \
    /usr/bin/apt-get install -qqy uwsgi uwsgi-plugin-python && \
    /usr/bin/apt-get clean && \
    /bin/rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

ADD /web-app.ini /etc/uwsgi/apps-enabled/

RUN /bin/mkdir /opt/web-app && \
    cd /opt/web-app && \
    /usr/bin/git clone https://github.com/application.git /opt/web-app && \
    /usr/bin/pip install -r requirements.txt && \
    /bin/chown -R www-data:www-data /opt/web-app

CMD ["uwsgi", "--ini", "/etc/uwsgi/apps-enabled/web-app.ini"]
```

Like in the Nginx Dockerile, we’re using ADD to copy the uWSGI config file into the image.

Going Multi-Container: Services

Dockerfile

```
FROM ubuntu:16.04

RUN /usr/bin/apt-get update -q && \
    /usr/bin/apt-get install -qqy build-essential git && \
    /usr/bin/apt-get install -qqy python-pip python-dev && \
    /usr/bin/apt-get install -qqy uwsgi uwsgi-plugin-python && \
    /usr/bin/apt-get clean && \
    /bin/rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

ADD /web-app.ini /etc/uwsgi/apps-enabled/

RUN /bin/mkdir /opt/web-app && \
    cd /opt/web-app && \
    /usr/bin/git clone https://github.com/application.git /opt/web-app && \
    /usr/bin/pip install -r requirements.txt && \
    /bin/chown -R www-data:www-data /opt/web-app

CMD ["uwsgi", "--ini", "/etc/uwsgi/apps-enabled/web-app.ini"]
```

The second RUN command creates the application directory, loads the code from GitHub, installs all Python package requirements and then sets permissions on the directory.

Going Multi-Container: Services

Dockerfile

```
FROM ubuntu:16.04

RUN /usr/bin/apt-get update -q && \
    /usr/bin/apt-get install -qqy build-essential git && \
    /usr/bin/apt-get install -qqy python-pip python-dev && \
    /usr/bin/apt-get install -qqy uwsgi uwsgi-plugin-python && \
    /usr/bin/apt-get clean && \
    /bin/rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

ADD /web-app.ini /etc/uwsgi/apps-enabled/

RUN /bin/mkdir /opt/web-app && \
    cd /opt/web-app && \
    /usr/bin/git clone https://github.com/application.git /opt/web-app && \
    /usr/bin/pip install -r requirements.txt && \
    /bin/chown -R www-data:www-data /opt/web-app

CMD [ "uwsgi", "--ini", "/etc/uwsgi/apps-enabled/web-app.ini" ]
```

This last line is something you normally find in all Dockerfiles. The CMD statement is the command that will be executed when the container first runs.

For our Flask application, we want to launch the uWSGI server, which is configured by our .ini file to serve the traffic over port 5000.

Going Multi-Container: Env Vars

```
version: ‘3'

services:
  mysql:
    . . .
    environment:
      MYSQL_ROOT_PASSWORD: ‘supersecure'
      MYSQL_DATABASE: 'mydb'
      MYSQL_USER: 'bryson'
      MYSQL_PASSWORD: ‘alsosecure’
  . . .

. . .
```

Before when we were using environment variables with our compose file, we listed them out individually with their values under the service.

Going Multi-Container: Env Vars

```
version: '3'

services:
  mysql:
    . . .
    environment:
      MYSQL_ROOT_PASSWORD:
      MYSQL_DATABASE:
      MYSQL_USER:
      MYSQL_PASSWORD:
  . . .

. . .
```

If you omit the values, docker compose will read the values in from the shell environment you are running the command in.

Going Multi-Container: Env File

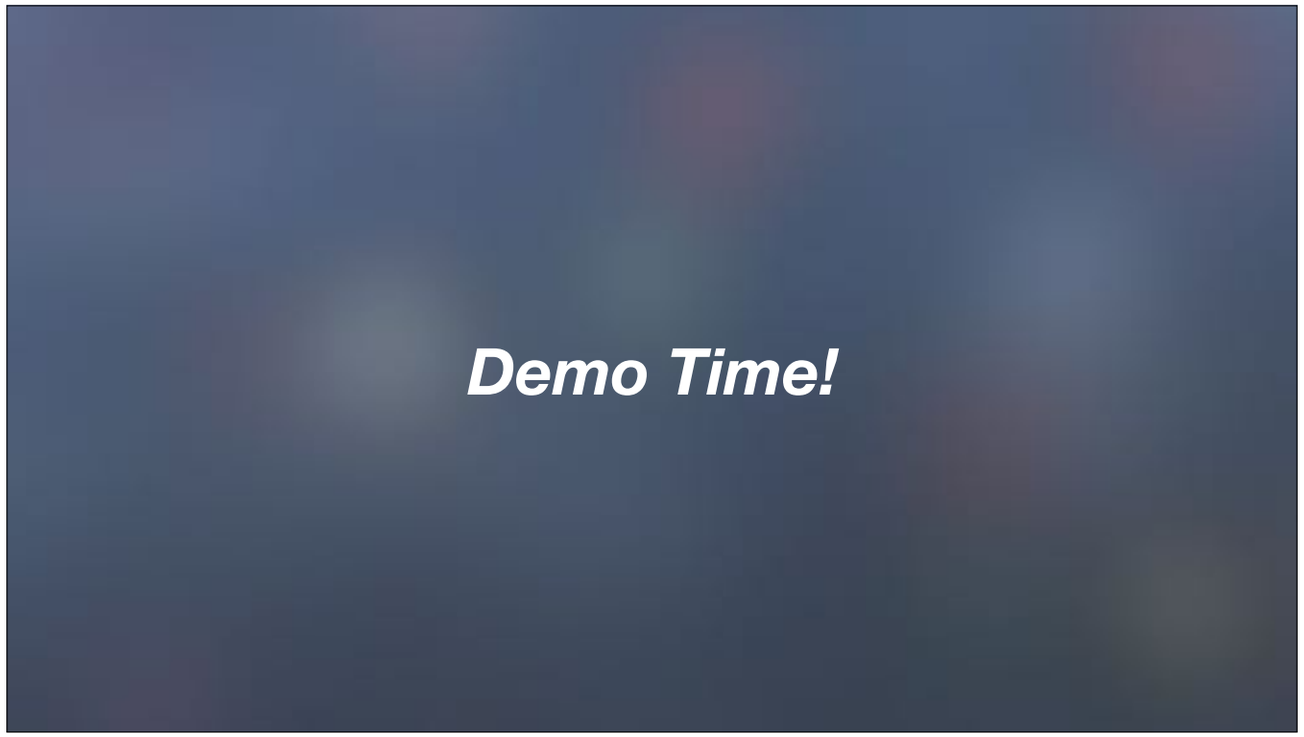
```
version: ‘3'

services:
  mysql:
    . . .
    env_file:
      - ./my_env_vars.txt
  . . .

. . .
```

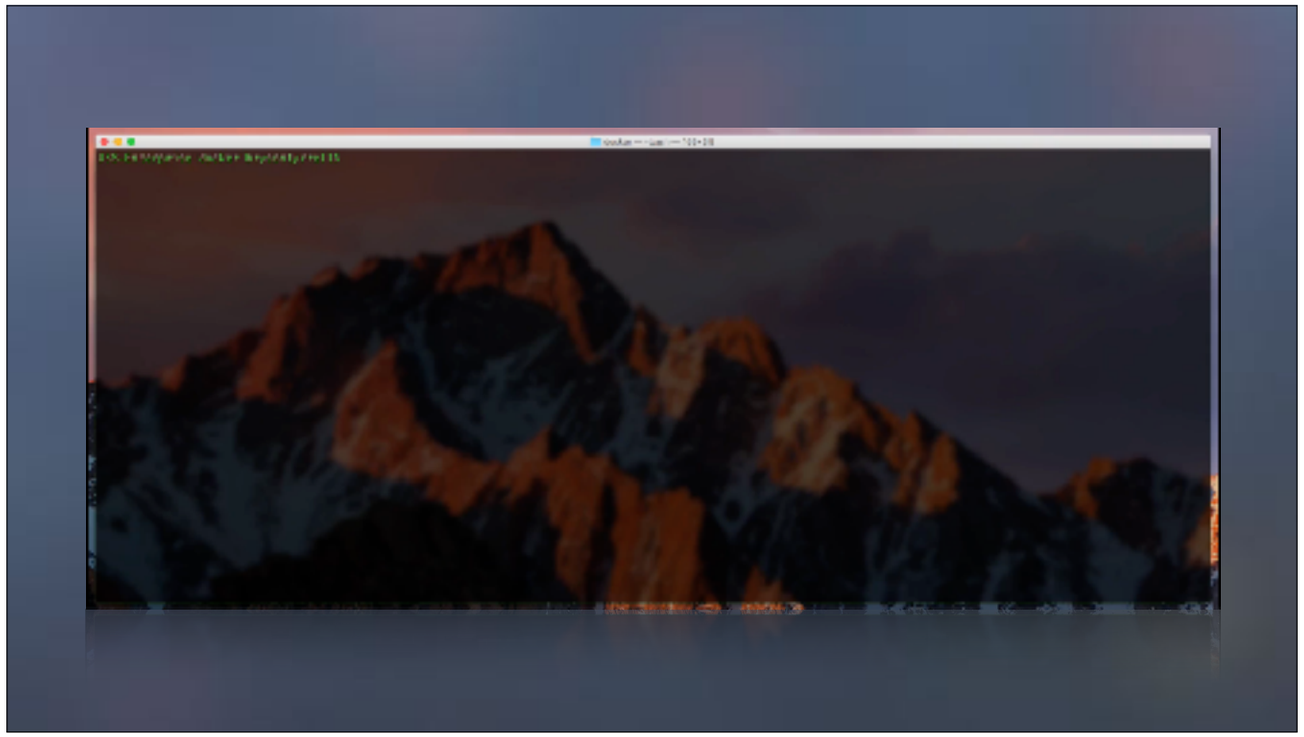
A third options is to store all the environment variables into a file in the directory and point to it using the ‘env_file’ key.

Going Multi-Container: Env File

```
version: '3'

services:
  mysql:
    . . .
    env_file:
      - ./my_env_vars.txt
  web:
    . . .
    env_file:
      - ./my_env_vars.txt
  . . .

. . .
```

This is a really useful feature if you need to share environment variables across multiple containers. Not only can you have more than one env file listed, but you can list the same one in multiple services.

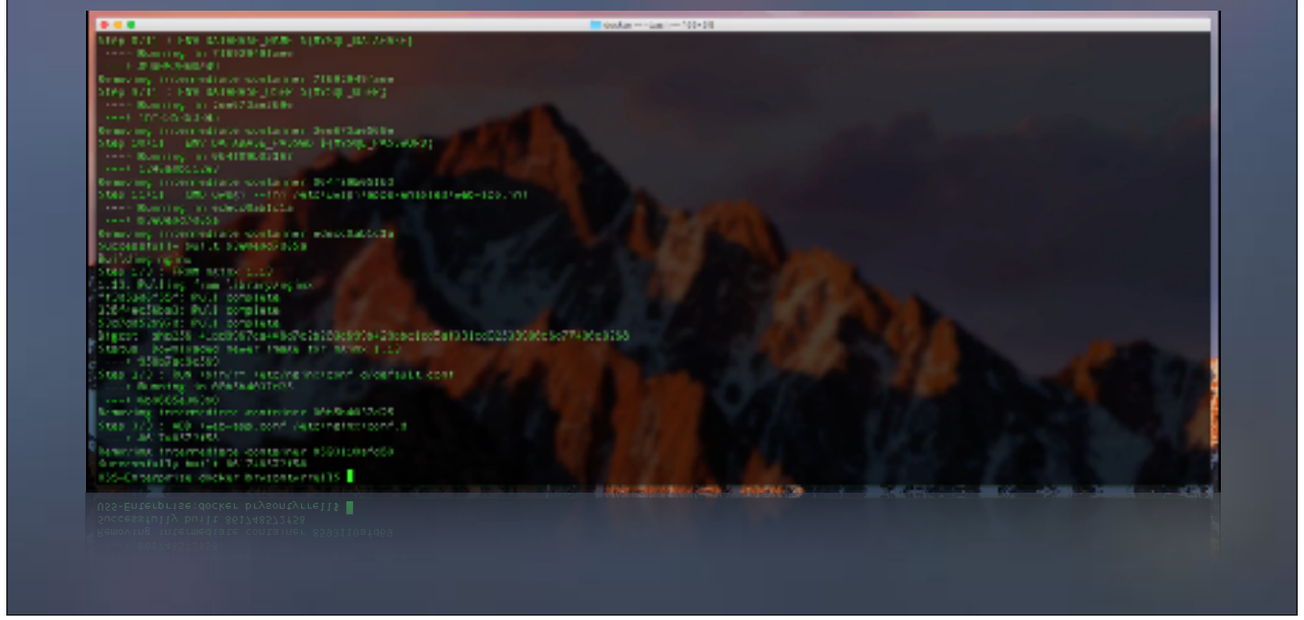

Demo Time!



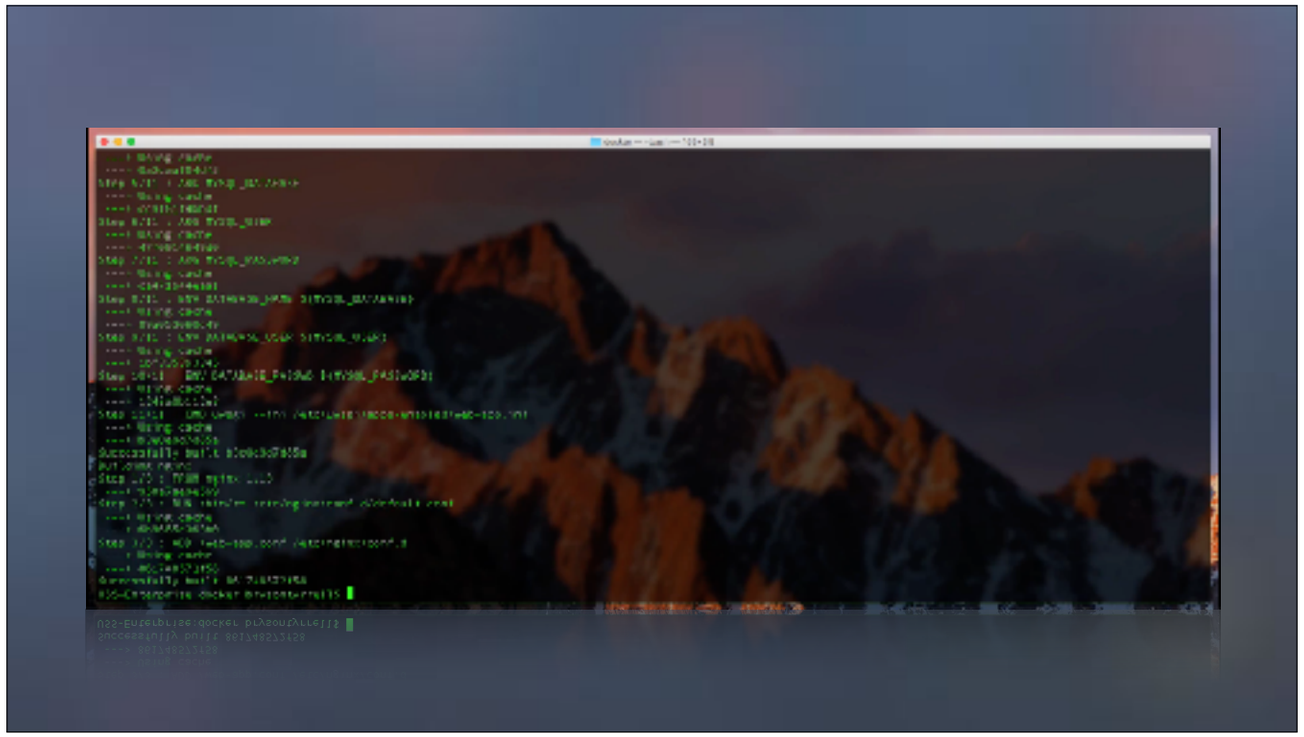
I’m still connected to the same Docker host and removed all the running containers.

I’m going to tell docker compose to build all the images in my compose file without running them. When you use docker compose up it will attempt to build the images if they are not already created. The build command can be used to create the images only.

This is sped up so we don’t have to wait through all the installs.

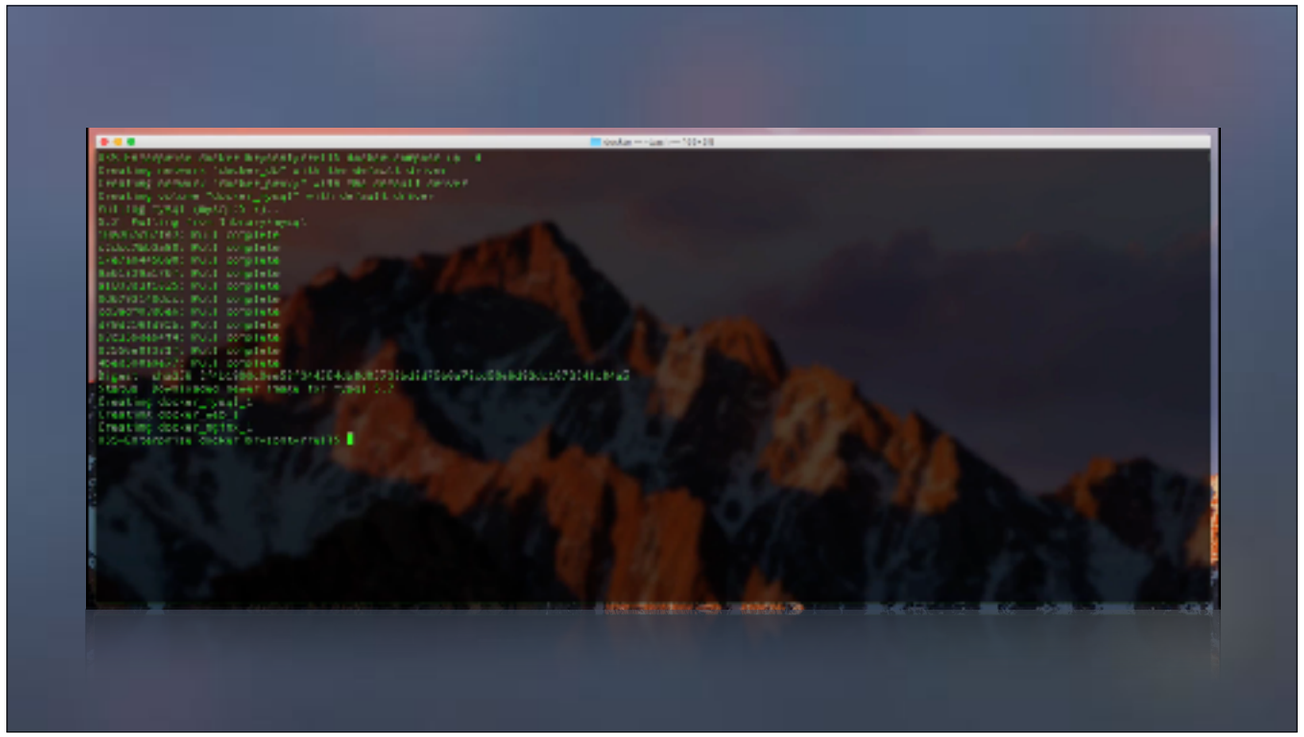


Just to show you, I’m running docker compose build again and the images do not go through the same build process.



Now when I run docker compose up we see the creation of the networks and volume first. When more than one item is created as a part of a compose file they are all prefixed with the name of the parent directory by default.

The mysql image is also being pulled down from Docker Hub. Any service in the compose file that doesn't reference a Dockerfile isn't included with the build command. Those services will be pulled down the first time docker compose up is run.



Now we can use the docker command to list out the running containers, the created networks and the volume.

What about deployments?

Reduced complexity by:

- Making testing easier ✓
- Making testing consistent ✓
- Making deployments simpler ?

Now, what about deployments? Docker Compose made my testing easier, and it made my testing consistent, but how does any of that work up until now lead into simpler deployments?

What about deployments?

```
version: '3'

services:
  mysql:
    image: mysql:5.7
    volumes:
      - /mysql:/var/lib/mysql
    restart: always
    env_file:
      ./my_env_vars.txt
    networks:
      - db

  . . .
```

```
. . .

web:
  build: ./web
  restart: always
  env_file:
    ./my_env_vars.txt
  networks:
    - proxy
    - db
  depends_on:
    - mysql

. . .
```

```
. . .

nginx:
  build: ./nginx
  restart: always
  ports:
    - "80:80"
    - "443:443"
  networks:
    - proxy
  depends_on:
    - web

networks:
  proxy:
  db:

volumes:
  mysql:
```

Didn't it, though? Here is the entire Compose file that we just went through. My application is comprised of three services that are are detailed here. They're set at fixed major versions here and in the Dockerfiles so rebuilding images on another host with this Compose file will yield the same setup.

What about deployments?

```
version: '3'

services:
  mysql:
    image: mysql:5.7
    volumes:
      - /mysql:/var/lib/mysql
    restart: always
    env_file:
      /my_env_vars.txt
    networks:
      - db

  . . .
```

```
  . . .

  web:
    build: ./web
    restart: always
    env_file:
      /my_env_vars.txt
    networks:
      - proxy
      - db
    depends_on:
      - mysql

  . . .
```

```
  . . .

  nginx:
    build: ./nginx
    restart: always
    ports:
      - "80:80"
      - "443:443"
    networks:
      - proxy
    depends_on:
      - web

networks:
  proxy:
  db:

volumes:
  mysql:
```

Configuring MySQL and the web app is done through environment variables that are passed by a file on the system doing the deployment. The difference between testing and production here is going to be different values between the two.

What about deployments?

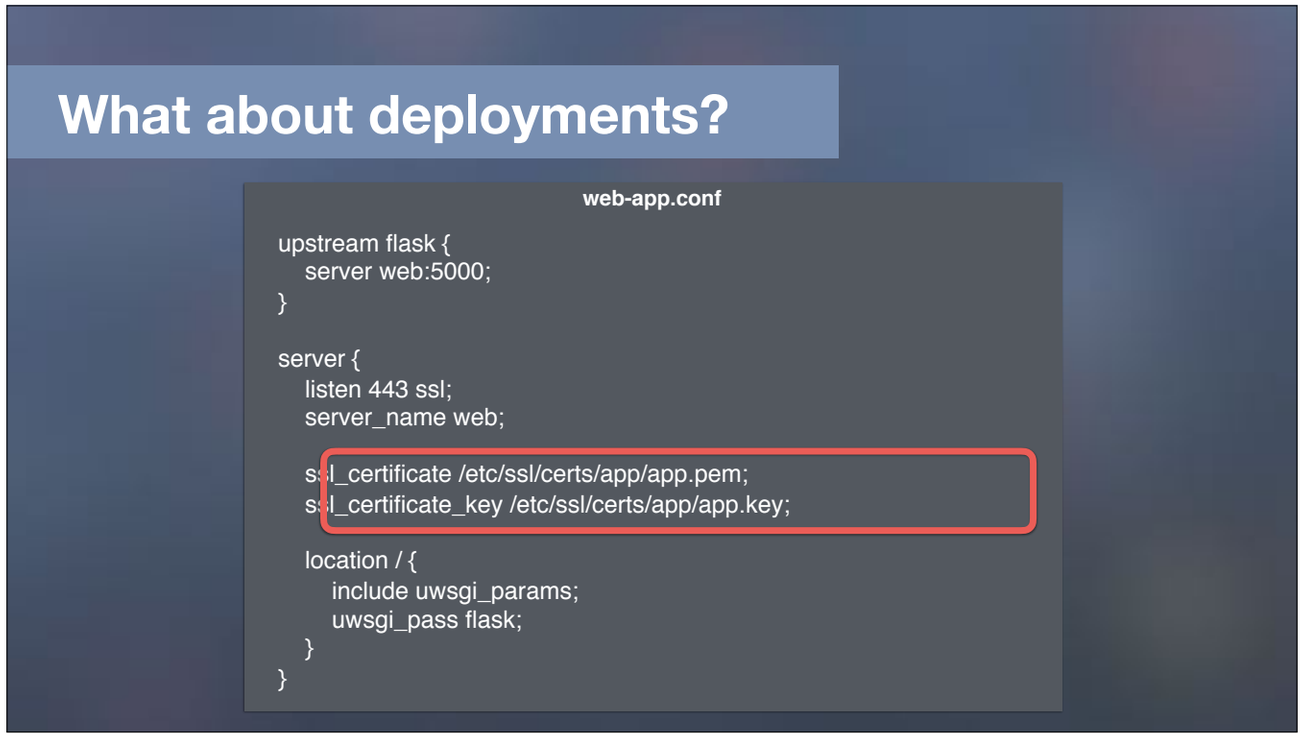
```
version: '3'

services:
  mysql:
    image: mysql:5.7
    volumes:
      - /mysql:/var/lib/mysql
    restart: always
    env_file:
      ./my_env_vars.txt
    networks:
      - db

  . . .
```

```
. . .

web:
  build: ./web
  restart: always
  env_file:
    ./my_env_vars.txt
  networks:
    - proxy
    - db
  depends_on:
    - mysql

. . .
```

```
. . .

nginx:
  build: ./nginx
  restart: always
  ports:
    - "80:80"
    - "443:443"
  networks:
    - proxy
  depends_on:
    - web

networks:
  proxy:
  db:

volumes:
  mysql:
```

Nginx only needs a few changes for a production environment - but these are changes that I can apply to my testing still. And those changes are all about TLS.

Never, ever deploy without a valid cert on your web app.



At a minimum, there isn't much that changes with Nginx to serve up HTTPS with a certiicate. You add these two entries to point to the public cert and the private key.

How the certificate and key are made available can vary. A not so great idea is having them copied into your Docker image by the Dockerfile. The reason being is that you will have committed your private key into one of those layers and if someone gets your image they now have the key.

What about deployments?

```
. . .

nginx:
  build: ./nginx
  volumes:
    - /etc/ssl/certs/app:/etc/ssl/certs/app
  restart: always
  ports:
    - "80:80"
    - “443:443"
  networks:
    - proxy
  depends_on:
    - web

. . .
```

My approach is to place the files onto the host in and ‘/etc/ssl/certs/app’ directory as shown and then mount that host volume into the container. Now I can have them securely loaded onto my Docker host and still have the container be able to access them when it loads.

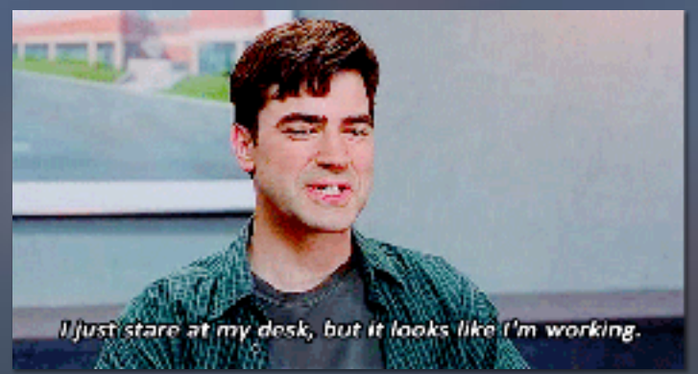
In testing I can keep this behavior by using a dedicated wildcard cert for a test domain or by using a self-signed certificate for the host.

What about deployments?

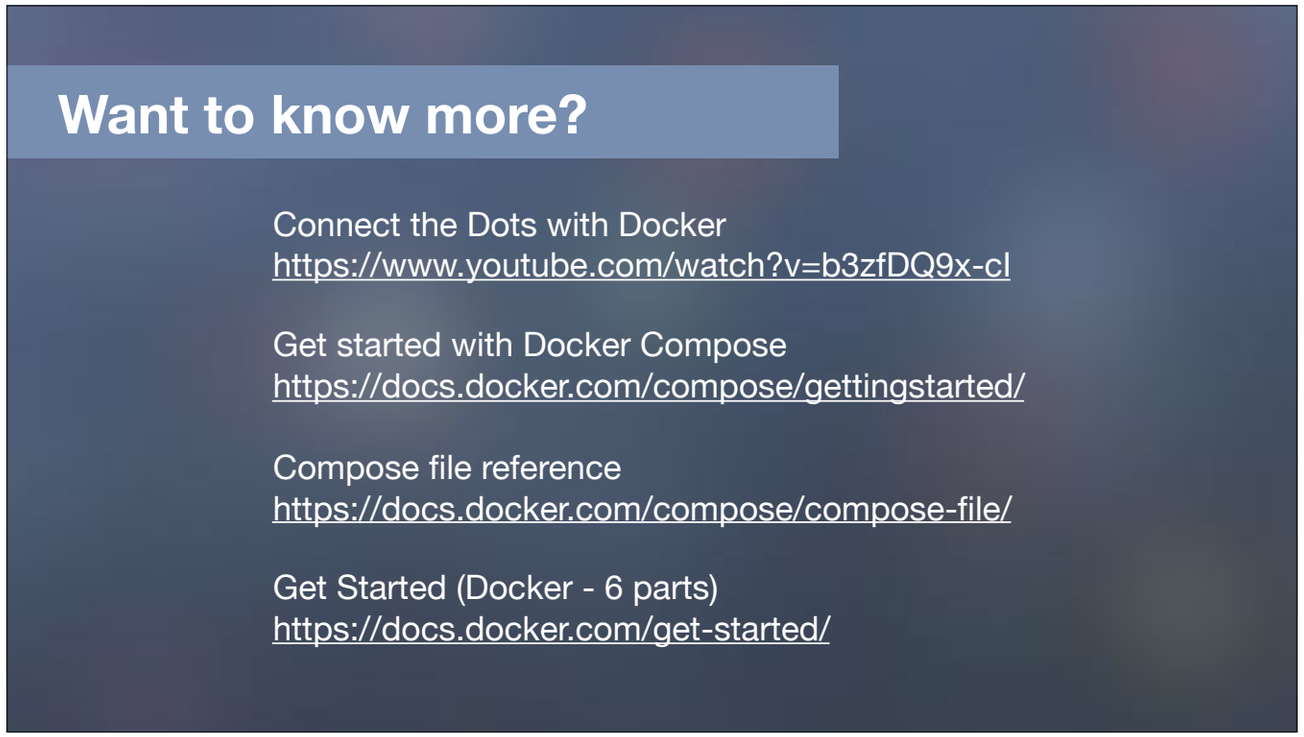
How to deploy my app:

1. Create and connect to a new Docker host
2. Clone the application repository
3. Copy the env, conf and certificate files
4. Run docker-compose build
5. Run docker-compose up -d

In the end I deployed these services internally on Docker hosts, VMs running in vSphere, as containerized apps. The amount of documentation I had to maintain for others to be able to redeploy one of those apps pretty much distilled down to:



And life just got so much easier.



Want to know more? Again, there's Pepijn's presentation from 2015. Go check it out.

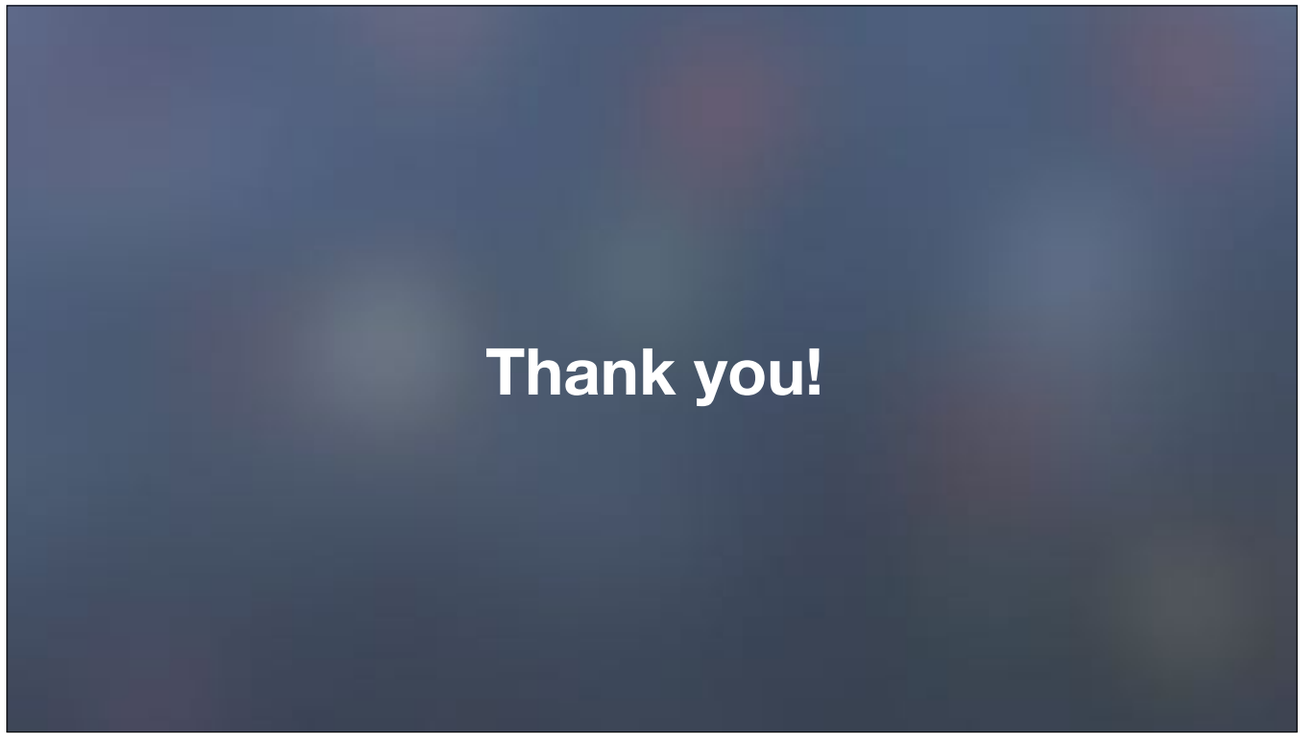
The "Getting started…" tutorial on Docker's site is a refresher on a a lot of what I covered here on depoying a simple Flask app with a Redis server.

To find out all the cool stuff you can do with Docker Compose, the file reference is a great place to poke around. Something to keep in mind is that these files are versioned so be sure you're looking at the correct reference.

If you want to go real deep on Docker, their "Get Started" guide is very comprehensive and even gets into Docker Swarm which I didn't touch on.

Q&A

bit.ly/psumac2017-153


Thank you!