Containers Without the Magic

Vince Salvino

@vincesalvino

slides: coderedcorp.com/resources

PyOhio 2018

What is a Container?

Common Answer = Dark Magic

Look how “easy” it is to use containers…

```
$ apt-get install docker
$ docker run hello-world
```

Real Answer = Bundle

- Bundle ALL software and system dependencies of your app
- **Less overhead** and complexity than a virtual machine
- **More control** than a requirements.txt

That’s nice… so why should I care?

Running your Python app “normally”

```
$ pip install -r requirements.txt

$ python myapp.py
```

- But wait… which version of python am I using?
- But wait… something in my requirements.txt conflicts with a different version of something else on the system.

And then the gods created virtualenv

```
$ virtualenv myapp
(myapp)$ pip install -r requirements.txt
(myapp)$ python myapp.py
```

- Now I can run different versions of python in each virtualenv!
- Now I can run different versions of EVERYTHING in my requirements.txt in separate virtualenvs!

But the gods were still not pleased

- My system only comes with python 2.7 and 3.2
  - Live with it.
  - Compile a different version of python for your system.
  - Install a sketchy binary or PPA from some random dude on the internet.

- My app needs a SYSTEM library installed, that is outside the scope of pip
  - Well on fedora you need to install [package]
  - On ubuntu you can install [package], but that version has a bug that doesn’t work with our app
  - Fools! - your app should only be pure python!

Some tried to please the gods by sacrificing resources to virtual machines

- But nobody wants that mess
- Why have we resorted to creating an entire OS image just to run our app?
- Bloatware, USA – now my nice lean app needs multiple gigabytes of memory and a 20GB disk just to run a copy of the OS and all those system libraries.

So the gods created containers

- *Actually, the concept of containers has existed for a long time (BSD jails - but only heathens use BSD)*

- Containers essentially let you specify a collection of system packages, code, files, etc. and run that all natively on the OS.
- It's like virtualenv for your entire OS!!!

[recap]

That’s nice…
so why should I care?

- If you ever needed a different version of python…
- If you ever had trouble installing a system dependency…
- If you ever needed to install your app on multiple systems (or multiple apps on one system) and found it involving a lot of tedium…
- If you want to easily distribute a fully working version of your app to others…

…then you might care about containers.

That’s cool…

How do containers actually work?

Container Tech Comparison

virtualenv

- A very basic containerization system
- Specifically for python
- Only handles python packages

```
# requirements.txt

Django==1.11

wagtail==2.0.1

mysqlclient
```

Container Tech Comparison

What if we had virtualenv for the whole system?!?!

- Install python versions
- Manage apache/system dependencies

```
# super requirements.txt

Django==1.11

wagtail==2.0.1

mysqlclient

Apache==2.4

Python==3.6

mod_wsgi

mod_redirect

imagemagick
```

… we do! It’s called LXC or Docker

Container Tech Comparison

LCX

- Linux Containers
- Starts from a base image which is like a lightweight mini-distro (ubuntu, etc.)
- Runs all the libraries and code of the mini-OS natively using the host's kernel.
- Similar experience to a VM, but much lighter and not actually virtualized.

Docker

- Very portable (Windows, Mac, Linux, cloud-native)
- Images only include the exact software you specify.
- Other software dependencies are handled by docker behind the scenes.
- Runs only the libraries and code you specify directly on the host.

Thinking in terms of a LAMP stack

- Start with a pre-defined OS image (Debian 8)

Thinking in terms of a LAMP stack

- Start with a pre-defined OS image (Debian 8)
- Install my app's system dependencies (e.g. imagemagik, libmysqlclient-dev)

Thinking in terms of a LAMP stack

- Start with a pre-defined OS image (Debian 8)
- Install my app’s system dependencies (e.g. imagemagik, libmysqlclient-dev)
- Install the version(s) of python from OS (2.7 and 3.4)

Thinking in terms of a LAMP stack

- Start with a pre-defined OS image (Debian 8)
- Install my app's system dependencies (e.g. imagemagik, libmysqlclient-dev)
- Install the version(s) of python from OS (2.7 and 3.4)
- Install virtualenv so I can run more than one python app

Thinking in terms of a LAMP stack

- Start with a pre-defined OS image (Debian 8)
- Install my app’s system dependencies (e.g. imagemagik, libmysqlclient-dev)
- Install the version(s) of python from OS (2.7 and 3.4)
- Install virtualenv so I can run more than one python app
- Install the version of Apache from OS (Apache 2.4.12)

Thinking in terms of a LAMP stack

- Start with a pre-defined OS image (Debian 8)
- Install my app's system dependencies (e.g. imagemagik, libmysqlclient-dev)
- Install the version(s) of python from OS (2.7 and 3.4)
- Install virtualenv so I can run more than one python app
- Install the version of Apache from OS (Apache 2.4.12)
- Set up a wsgi, jump through hoops, create a virtualenv, set things to start on boot, etc.

Thinking in terms of a LAMP stack

- Start with a pre-defined OS image (Debian 8)
- Install my app’s system dependencies (e.g. imagemagik, libmysqlclient-dev)
- Install the version(s) of python from OS (2.7 and 3.4)
- Install virtualenv so I can run more than one python app
- Install the version of Apache from OS (Apache 2.4.12)
- Set up a wsgi, jump through hoops, create a virtualenv, set things to start on boot, etc.
- **Copy my code to the server and restart Apache**

Thinking in terms of Docker

- Pre-define a docker image that includes apache, python, and other system dependencies.

Thinking in terms of Docker

- Pre-define a docker image that includes apache, python, and other system dependencies.
- Pre-load my code in the docker image.

Thinking in terms of Docker

- Pre-define a docker image that includes apache, python, and other system dependencies.
- Pre-load my code in the docker image.
- Pre-define what to do when it starts (e.g. start apache)

Thinking in terms of Docker

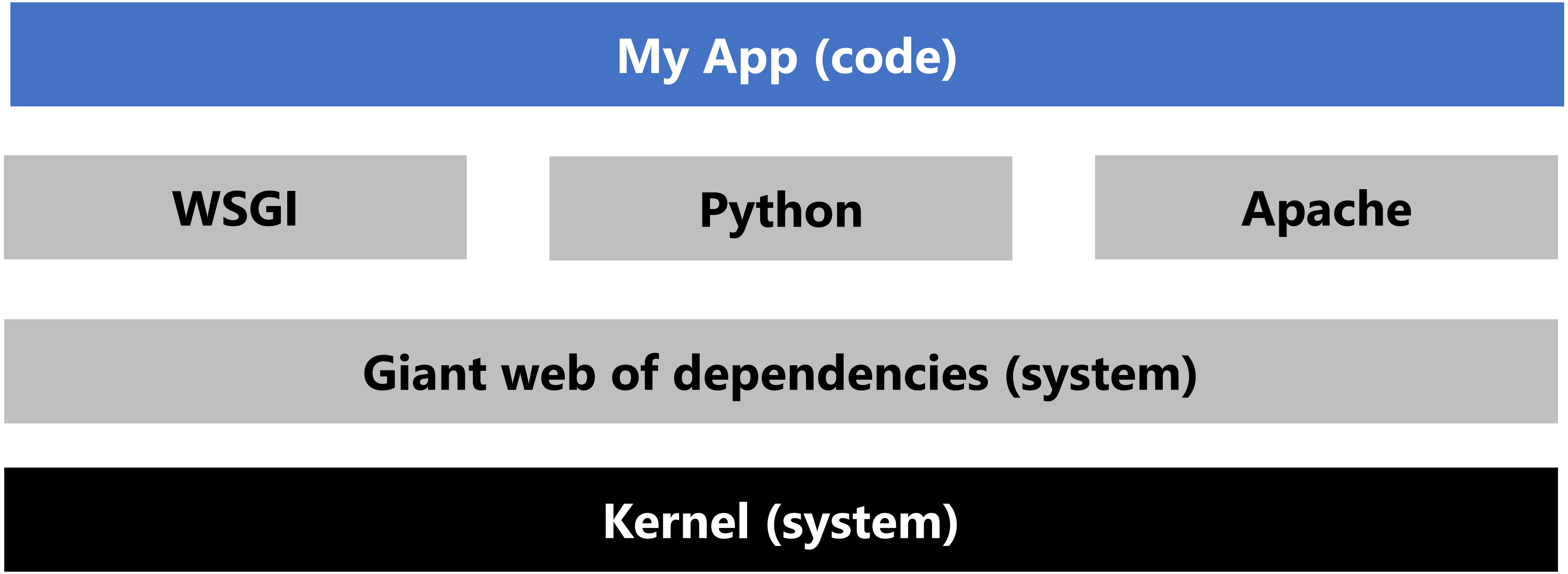
- Pre-define a docker image that includes apache, python, and other system dependencies.
- Pre-load my code in the docker image.
- Pre-define what to do when it starts (e.g. start apache)
- **Run the docker image**

Thinking in terms of Docker

- Pre-define a docker image that includes apache, python, and other system dependencies.
- Pre-load my code in the docker image.
- Pre-define what to do when it starts (e.g. start apache)
- **Run the docker image**

My docker image will now run EXACTLY the same on any OS because all dependencies and files have been pre-defined and are managed by docker, instead of being managed by the OS or manually by the sys admin.

Visualized LAMP stack



My App (code)

WSGI

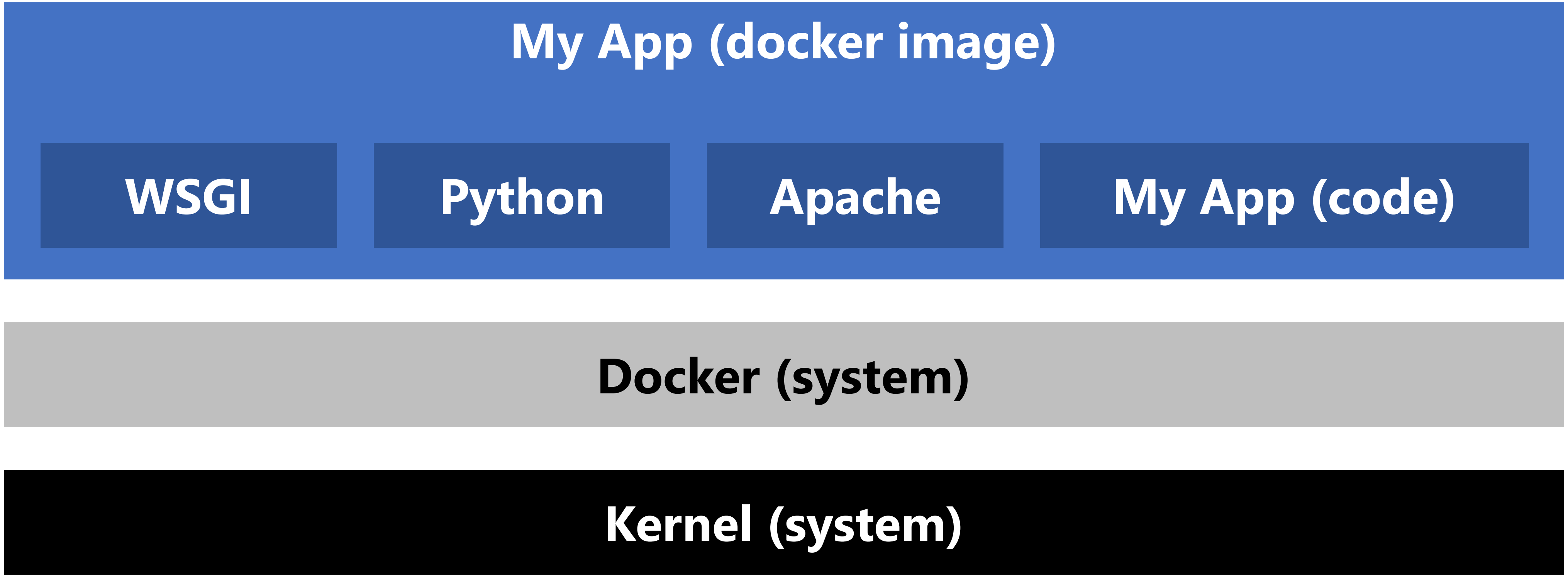
Python

Apache

Giant web of dependencies (system)

Kernel (system)

Visualized Docker stack



Quick Start

Let’s Dockerize your Python app

My app looks like this:

```
/myapp/
    myapp.py
    requirements.txt
```

Let’s Dockerize your Python app

Now add a Dockerfile. Think of the dockerfile as a requirements.txt for your entire system.

```
/myapp/
    myapp.py
    requirements.txt
    Dockerfile
```

Let’s Dockerize your Python app

```
# Dockerfile

FROM python:3.6

COPY . /code/
RUN pip install -r /code/requirements.txt

CMD python /code/myapp.py
```

Let’s Dockerize your Python app

```
# Dockerfile

FROM python:3.6

COPY . /code/
RUN pip install -r /code/re

CMD python /code/myapp.py
```

This is a simple example, using the actual python file as the final CMD command.

For something like a LAMP stack, this Dockerfile would include installation of apache and dependencies, and the final CMD command would probably be to start apache.

Let’s Dockerize your Python app

We just created a Dockerfile that defines everything our app needs, and what to execute

Now we build a docker image of our app

```
/myapp $ docker build -t myapp_image
```

Let’s Dockerize your Python app

```
/myapp $ docker build -t myapp_image
```

- This one command provisions all dependencies we defined for our app, and packages it up into a single container image.

- This image is a binary distributable. Think of it like an “.exe” that contains our app and everything our app needs, and tells the system what to execute.

It’s all dockerized

Now we have a docker image called **myapp_image**

Run your Python app

Now we can create and run an actual container (instance of our app) from the docker image

```
$ docker run myapp_image
```

Remember that dark magic from the first slide...

Hopefully now it makes a little bit of sense.

```
$ apt-get install docker
$ docker run hello-world
```

Avoiding
“New Shiny Syndrome”

Containers provide a way of bundling code AND system dependencies into one binary

When to USE containers

- App runs on multiple systems
- Multiple different apps run on one system.
- Easily distribute a fully working app to other systems or users.

When NOT TO USE containers

- One app per system
- System dependencies do not need upgraded or changed frequently
- The app does not get distributed to other systems or users.

Let’s Talk

@vincesalvino

slides: coderedcorp.com/resources

salvino@coderedcorp.com