Intro to Docker and Remote Python Debugging

or

How to Throw Your Code Into a Container, Throw it Overboard, and Still be Able to Interact With It

Say Hwat Now?

What is Docker, and Why do I Care?

It's kind of like virtualization on steroids

Full Virtualization is Expensive

In order to create a fully virtualized environment, you must fake each piece of hardware and the interactions with it. This comes at a cost -- the host machine (the one running a "real" or host OS) has to go through the trouble of emulating all of this.

This is how VirtualBox and VMWare work -- the underlying system might have no idea that it's actually being virtualized and expect to interact with real hardware.

Paravirtualization is Cheaper

In paravirtualization, the VM uses a modified kernel. This kernel knows how to interact with the underlying system and asks it directly for resources (like memory or IO) directly via a hypervisor rather than to issue a system call.

This greatly reduces the amount of virtualization needed, and thus reduces the overhead of running VMs.

Containerization is About as Cheap as You Can Get

The Linux kernel has a feature called LXC. Essentially, the VMs are not really VMs anymore -- they're just walled-off processes on the host machine. This means that there's nothing to "fake" or to virtualize. It also means that the host system has fine-grained control over what those "containers" (process groups) can do. CPU, IO, and memory usage can be easily controlled and regulated.

Ok, Containers are Cheap to Run...So What?

They're also cheap to develop and deploy.

Each container is made up of "layers". You can start with someone else's Docker image and add your own stuff on top. When you have an update to roll out, you're really just adding another layer.

Most cloud services also directly support deploying contianers. No SSH'ing in and manually installing stuff -- installation

My Head Hurts

Ok, let's get into some concrete stuff.

Let's introduce the `Dockerfile`. This is a file that defines how to build a container. Let's look at a simple example

Dockerfile

```
FROM python:3.6
MAINTAINER Brian Stempin "brian.stempin@gmail.com"
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Dockerfile Breakdown

```
FROM python:3.6
```

This line declares that this is defines additional layers on some other existing image. By default, Docker will reach out to a public repo to look for this image. You can host your own private repo and images if you wished. This one happens to be public.

Dockerfile Breakdown

```
MAINTAINER Brian Stempin "brian.stempin@gmail.com"
```

This just sets some metadata on the image. It doesn't really impact much.

Dockerfile Breakdown

```
COPY . /app
```

This tells Docker to copy all of the files in the current directory into the `/app` directory in the container. The container has it's own file system. There are even ways to set up sharing between the host and container.

Dockerfile Breakdown

```
WORKDIR /app
```

Any other commands executed in the container will happen in /app

Dockerfile Breakdown

```
RUN pip install -r requirements.txt
```

This command assumes we have some requirements in our current directory and will ask `pip` to install them.

Dockerfile Breakdown

```
ENTRYPOINT ["python"]
```

If we run this container as an executable (Yeah -- containers don't have to be daemon-like -- they can start, do work, make output, and go away.), it will run `python` by default.

Dockerfile Breakdown

```
CMD ["app.py"]
```

This sets defaults. So, if you execute your container without passing it a command, it will run `python app.py`. You could have it run other stuff via CLI arguments.

Try it Out

From within this presentation's repo:

```
cd example_1
docker build -t test_container .
docker run --rm test_container
```

What Did I Just Do?

1. You told Docker to build a container, defined by the Dockerfile in `.` and to name it `test_container`. Normally, it's `name:version`, so it will assume this is `test_container:latest`
2. You told Docker to execute the container, expecting the container to complete, and to clean up any resources that container used (the `--rm`).

Common Uses

1. You can use containers like big, portable executables. Once the images are downloaded, you're good to go -- nothing to install!
2. You can run long-running processes and treat them like VMs. You coult run a `Flask` or `Django` app in a container.

Intermission and Questions Before Moving on to Python Stuffs

That was a lot, and this wasn't comprehensive at all. It's enough to help you understand documentation and common use cases/patterns.

Python and Docker

You can run Python code in containers!

You can ship your app as a container to a cloud for deployment or to a customer. Instead of sending along a set of installation instructions, you send along the container image layers and instructions on environment variables or commands the container understands.

That Poses Challenges

Containers are their own processes, so sometimes things will work locally that won't work in a container and vice versa.

How do you do testing and debugging within a container?

The containers is, in some contexts, a remote environment. The rest of this presentation is an intro to remote debugging in Python and how to get it to work in Docker.

Python Remote Debugging 101

- There are a bunch of debugging tools out there. `pdb` comes with Python. Most of your IDEs also have debuggers.
- I'm a PyCharm user, so the rest of this tutorial will focus on PyCharm
- I promise I'm not being selfish
- One of the statements above is a lie

PyCharm Remote Debugging Tools

Long story short, there are two modes: remote interpreter and Python debug server

PyCharm: Remote Interpreter

Pros:

- Requires a remote deployment
- Requires an SSH connection
- Simple

Cons:

- Most containers don't run an SSH service

PyCharm: Debug Server

Pros:

- Works without modifying the container

Cons:

- Requires external dependencies
- Requires code changes
- Makes my head hurt
- I'm lazy

They Both Kind Of Suck

Isn't there something more seamless?

PyCharm: Docker as a Remote Interpreter

General steps:

- Build and name a Docker image
- Tell PyCharm to use this as the project's interpreter
- ???
- Profit

Demonstration

Pros and Cons of This Approach

Pros:

- Minimal setup
- No manual modification of your Dockerfile

Cons:

- You will have to rebuild your container often, which can take time

Can we do Better?

Yes, but not in this demonstration.

PyCharm also supports `docker-compose`, which allows you to:

» Run multiple containers, like a dev database

» Define parameters for running containers, like passwords and ports

» Just magic in general

Where to From Here?

Docker and Docker-Compose are wonderful tools with large communities. There are plenty of guides on their respective websites.

If you're running Windows, be prepared for an up-hill batter if you are trying to install and use Docker.

If you're on Linux or OS X, getting these tools will be easy-peasy.