Containers :: Docker

By Benjamin Brewster and Elijah Voigt

Please boot up the "CentOS_CLI_Reference" VM in VirtualBox now

Why You Need to Care

- Because someday you'll have to:
  - Run a ton of programs all at once
  - Run a complicated, multi-process system, where taking the whole thing down at once to upgrade or repair part of it is unfeasible
  - Work on a system whose functionality needs to be ultraportable and/or ephemeral
  - Want to download fully functional, no-configuration-needed instances of WordPress, etcd., Apache, Nginx, MongoDB, MySQL, RabbitMQ, Python, or a ton of other complicated programs

Why You Need to Care, In Context

- The containers we’re going to talk about are something that will get you jobs, but unless you’re running a really big system, you’re less-likely to use in a *small* production environment
- But for development, research, practice, dangerous trials, training, and being on the cutting edge, you *need* to know what they are and how to use them
- They’re a now defacto-required devops tool

Containers

- Containers are a form of process isolation
- These specially-encapsulated processes have no connection to other processes running on the same computer, except those network connections that you set up via port forwarding
- Each non-running container is essentially a zip file with its own filesystem, and metadata about the container
- Importantly, this bundled filesystem *contains the dependencies* needed to run the container

Containers - Apache Example

- E.g., an Apache container is a binary package of executables and files that contains:
  - The program you want to run (Apache, whose main process is `httpd`)
  - The parts of the OS (CentOS, say) that the program will use
  - The particular library dependencies needed by Apache (libxml, etc.)
- You can run this container, which starts up the Apache server in its own little sandbox, where it thinks it’s running on CentOS, all properly configured! Start it, stop it, start a million of them!
  - Doesn’t know about other processes running on host (and doesn't care)
  - Doesn’t know what the host’s OS is

Containers - Apache Example

- The Host runs an overseer program that knows how to feed the running Apache process the various services it might need
- Because the container lives to service Apache, there isn't a separate full general-purpose OS that's running just to sustain it - only enough of the OS is running as to enable Apache to run as if it was on a full OS install
- Containers don’t boot, no VM hypervisor starts them up, and no virtual disks are needed for their storage

Containers vs. VMs

- A VM:
  - Runs on emulated hardware, controlled by a hypervisor
  - Uses emulated drives: the VM has to fully (and slowly) boot from drive partitions, start the OS contained on another partition, then offer a login to the user
  - Uses disk drive files that are many (many) GB in size
  - Offers total process isolation: guest OS cannot access Host OS
- A Container:
  - Uses pieces of the Host OS (or pieces of another stored OS)
  - Doesn’t boot, use “init”, or any other OS start procedure
  - Is isolated from other processes, but *may* be able to access the Host OS at large if misconfigured
  - Starts and runs lightning quick

Containers - Why

- Again: containers are programs packaged with their dependencies, which makes deploying and upgrading them easy

- Why use containers?
    - Production engineers use them to get the most out of their hardware, as they can start as many services as possible - each it’s own container, which can be brought up and down, duplicated, configured, repaired
    - Developers use them because they *are* isolated from other dependencies, start and stop quickly, which makes iterative development in production-like environments easy

Containers - Key Points Summary & Usage

- Key point: you never log into a container!
- Containers are wrappers around a specific process that has access to the underlying OS running it
- Can be configured to be interactive with a shell or UI, if needed, but that’s very atypical usage

- **Critically, containers should be *ephemeral*: they are normally created and destroyed on the fly**
- **Don’t get into the habit of thinking that they are supposed to be kept around permanently like VMs, though they can be stopped and restarted**

Our Container Implementation: Docker

- We’ll be using Docker as our example
- Docker is the best (and now defacto) game in town

- The Docker daemon is called `dockerd`, which listens for commands coming in through it’s Docker API

- The client that interprets CLI commands and sends them to `dockerd` is the Docker client called `docker`

Our Container Implementation: Docker

- **Containers** are the instances of a Docker **Image**
    - i.e., **Images** are read-only templates used for distribution, and multiple **containers** can be created from them

- Containers cannot be distributed (they're ephemeral, remember); only images can be distributed

Installing Docker on CentOS7

Say	**Demonstrate**
• https://docs.docker.com/engine/installation/linux/docker-ce/centos/#install-using-the-repository	
• Install the prereqs, which are linux tools that support drive mapping and VMs	• `$ sudo yum install -y yum-utils device-mapper-persistent-data lvm2`
• Tell CentOS to use the Docker repository	• `$ sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo`
• Install Docker CE (Community Edition), aka just docker; answer ‘**y**’ on questions	• `$ sudo yum install -y docker-ce`
• Start docker	• `$ sudo systemctl start docker`
• Set docker to be running at system boot	• `$ sudo systemctl enable docker`
• Enable your user to run docker commands without using `sudo` by adding it to the Linux group	• `$ sudo usermod -aG docker centosuser`
• This enables the new group membership permissions	• Log out and back on

Test Docker

Say

- Is it running?
- Get the "hello-world" image, run it
- Read the hello world output

Demonstrate

- `$ docker info`
- `$ docker run hello-world`

Docker Commands - Pull

SAY

- Used to pull a Docker image down from a Docker **Registry**, which contains many repositories of images
- Docker clients connect to registries (which can be public or private) to push (upload) or pull (download) images
- A repository typically holds just one project, though many different versions can be in that same repository: "3.5", "latest", "temp", "testingbug123", "usethisfordemo", "nightly", etc.
- The default Registry is Docker Hub
  - https://hub.docker.com

- What's alpine? It's an extremely small, extremely fast Linux distro that is perfect for containers
- More on alpine in our next lecture

DEMONSTRATE

```
$ docker pull alpine:latest
```

```
$ docker pull alpine:3.5
```

Docker Commands - Images

Say

- Question: How many images have we pulled so far?
- Lists all the loaded images we’ve pulled

Demonstrate

- `3`
- `$ docker images`

Docker Commands - Run

Say

- Run is how we actually execute a container
- This command:
  1. Creates a new container from the latest "alpine" image
     - Because we didn't specify running an already-existing container, a new one is created
     - Since we didn't give this new container a name, a random one is assigned
     - Not specifying an alpine version means use the latest - here we get specific:
  2. Starts up the new container
  3. Runs the CLI command "cat /etc/os-release" inside the container
  4. Shuts down the container (which still persists, it's just not running anymore)

Demonstrate

- `$ docker run alpine cat /etc/os-release`
- `$ docker run alpine:3.5 cat /etc/os-release`

Docker Commands - ps

SAY

- Gives us info about existing containers
- By itself, gives us info about only currently running containers
- Run with the -a switch, we get information about all existing containers, including stopped ones
- Again: a new container is created from the image each time you run it, unless you specify the running of an already-existing container

DEMONSTRATE

- `$ docker ps`
- `$ docker ps -a`

Docker Commands - Run detached

Say

- Run a nginx webserver, alpine version, start it detached (a daemon, run until killed), and set up port forwarding so that port 80 can be reached from port 8080
- We just downloaded and started a web server running with one command!

Demonstrate

```
$ docker run --name web -d -p 8080:80 nginx:alpine
```

Say

- Get a web page from our server on port 8080
- See the running container

Demonstrate

```
$ curl localhost:8080
$ docker ps
```

Docker Commands - Stop, Restart, Stats

Say	**Demonstrate**
Shut down the web container	`$ docker stop web`
Verify that it is no longer running	`$ docker ps`
There it is, stopped	`$ docker ps -a`
Restart the web container with all the *same* commands and ports as when it was initially ran	`$ docker restart web`
Verify the ports are still set up	`$ docker port web`
Get that web page again	`$ curl localhost:8080`
Get stats on running containers; like `top`	`$ docker stats`
	`CTRL+C`
Shut down the web container	`$ docker stop web`

Docker Commands - Remove

Say	**Demonstrate**
• Let’s get rid of that container instance built from the older version of the alpine image	
• Display all containers	• `$ docker ps -a`
• Remove the older version container	• `$ docker container rm <random name of alpine:3.5>`
• Now, let’s get rid of the older version image itself	
• Here’s the image	• `$ docker images`
• Remove it • Note that the text returned talks about the “tag” 3.5 • Tags can hold a version number, sure, or a text label like “alpine” as we saw with nginx • When an image has no tags left, *that’s* what causes it to be removed • Images can be “copied” by giving them a tag - that other tag can be used to start that copy up	• `$ docker image rm alpine:3.5`
• Check images to see it’s gone	• `$ docker images`

Docker Commands - Remove All Exited

Say

- You're going to end up with a lot of old containers!
- Get a list of exited containers
  - -q flag means to only display the IDs
  - -f flag means to filter, where a state parameter (status, name, etc.) is given followed by one of:
    - created, restarting, running, removing, paused, exited, deaddocker
- Here's to remove them all at once: get the list of results from a subshell, then provide it to the standard remove command
- Look Ma, no containers
- The --rm flag automatically deletes the container after it terminates, **which is the container paradigm**
- See, no hello-world!

Demonstrate

- `docker ps -q -f status=exited`
- `$ docker container rm $(docker ps -q -f status=exited)`
- `$ docker ps -a`
- `$ docker run --rm hello-world`
- `$ docker ps -a`

Docker Commands - Interactive

Say

- Let's run a docker alpine container in interactive mode, so we get a CLI interface!
- Run commands!
- Can we get out of this container and leave it running? This is called detaching.
- Back in our VM
- See how it is still running?
- Here's how we (re)attach to that container
- Kill off the shell from within

Demonstrate

- `$ docker run -it alpine sh`
- `# ls -pla`
- `# ping www.google.com`
- `CTRL+P then CTRL+Q`
- `$ ls -pla`
- `$ docker ps`
- `$ docker attach <random cont name>`
- `# exit`

Getting Our Containers Somewhere Else - Repo

SAY

- First, we need a way to get our containers off of our local machine
- To do this, we'll use the default Docker registry, Docker Hub
- First, sign up for an account at docker.com (which is Docker Hub)
- From your docker hub dashboard, create a new private repository

DEMONSTRATE

- Log into hub.docker.com, show Dashboard

Getting Our Containers Somewhere Else - Push

Say	**Demonstrate**
• Remember that you don't push containers (instances of images) - you only push images (the templates)	
• Copy an image (this is just a clone of the existing one)	• `$ docker tag hello-world stonesand/brewpriv:hello-world`
• See the new clone	• `$ docker images`
• Login to Docker Hub, which **persists through reboot!**	• `$ docker login --username stonesand --password-stdin < ~/.docker/creds`
• Now upload the image to the repo that shares the same name	• `$ docker push stonesand/brewpriv:hello-world`
	• See it at docker.com->brewpriv->Tags
• Logout when you're ready	• `$ docker logout`

Getting Our Containers Somewhere Else - Pull

Say

- Do a test pull to make sure it uploaded correctly
- Logout when you're ready
- Connect to another machine that has Docker installed (note: OSU's servers don't have Docker installer)
- Initiate the same pull command

Demonstrate

- `$ docker pull stonesand/brewpriv:hello-world`
- `$ docker logout`

Conclusion

- Docker is an incredibly powerful tool to test out software and do Continuous Integration:
  - Safely
  - Quickly
  - Easily distributable

- Our next lecture will show how to build your own images!