Creating Containers in Alpine

Benjamin Brewster & Elijah Voigt

Please boot up the "CentOS_CLI_Reference" VM in VirtualBox now

Why You Need to Care

- Because someday you'll have to:
  - Build your own containers for a specialized task
  - Modify someone else's container
  - Create extremely small, fast yet complicated systems that are easier distributed in a complete form, than providing instructions to build

Alpine

- Alpine is an extremely small, extremely fast Linux distro that is perfect for containers
- The entire container download is only 4 MB!
- This is exactly what we want, since containers are acting as a form of package management: we want as little overhead as possible
- This gives us:
  - Faster-starting containers
  - Smaller containers, which improves both storage and distribution
- All of this encourages and enables experimentation

Working With Alpine

Say

- Lets pull the latest alpine image again
- We can see this in our main images list
- Let's spin it up interactively
  - `--name`: Name this container so it could be started back up again later
  - `-i`: Makes the container interactive
  - `-t`: Gives us a standard terminal to interact with
  - `i` and `t` are always used together, for our purposes
- Verify we're in the container

Demonstrate

- `$ docker pull alpine:latest`
- `$ docker images`
- `$ docker run -it --name alptest alpine sh`
- `# cat /etc/os-release`

Working With Alpine

Say

- This is the `sh` shell, it will use the sh prompt
- Acts like a shell, smells like a shell

Demonstrate

- `# ps`
- `# echo -e "junk\njunk2" > junkage`
- `# cat junkage`
- `# ping www.google.com`
- `^Z`
- `# jobs`
- `# fg %1`

Say

- alpine uses busybox to add common UNIX utilities
- Busybox versions are tiny in size, and have fewer options than the normal ones
- Most normal commands are actually just symlinks to the busybox binary(ies)

Demonstrate

- `# which ls`
- `# ls -pla /bin/ls`

Working With Alpine

SAY

- Instead of the standard glibc standard library, alpine uses musl, a much smaller version that still completely functions and produces smaller binaries

- gcc isn't installed by default, though, since alpine is focused on *running*, not *development*, but it can be installed like this:
  - Adds 175 MiB (183.5 MB) to our 4 MB distro

DEMONSTRATE

- `# apk -U add build-base gcc abuild binutils binutils-doc gcc-doc`
- `# vi hw.c`
- Add:

```
#include <stdio.h>
void main() { printf("Hello, World!\n"); }
```

- `# gcc -o hw hw.c`
- `# ./hw`

Building a Webserver in a Container - NodeJS

SAY

- Let's create a very simple webserver using NodeJS in an alpine container
- NodeJS is a wildly popular JavaScript runtime engine that runs JavaScript code *on the server*
- This is not the normal way of doing things: JS is traditionally a *client-side* scripting language for modifying a web page after it's downloaded to your browser: think menus, UIs, etc.
- If the entire *concept* of "JavaScript everywhere" bothers you, you're in good company

- Start with a clean copy of alpine with a name
- Install the NodeJS engine into our alpine shell
  - Adds 61 MiB (63.9 MB) to our 4 MB distro

DEMONSTRATE

- `$ docker run -it --name webtest alpine sh`
- `# apk -U add nodejs`

Building a Webserver in a Container - Code

Say

- Write the following code to /srv/server.js using vi:
- If you remember from CS344, I *told you* that at some point in your CS careers, you would be *required* to use vi because it was the only tool available to you: *that day has come!*

Demonstrate

- `# vi /srv/server.js`
- Type:

```
var http = require('http');
http.createServer(function (request, response)
{
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8080);
console.log('Server started');
```

Building a Webserver in a Container - Code

Say

- Write the following code to /srv/server.js using vi:
- If you remember from CS344, I *told you* that at some point in your CS careers, you would be *required* to use vi because it was the only tool available to you: *that day has come!*
- OR cheat:
  - Note that this adds 61 MiB (67.1 MB) to our 64.9 MB distro!
  - -L means follow the goo.gl redirection
- Test the curl

Demonstrate

- `# vi /srv/server.js`
- Type:

```
var http = require('http');
http.createServer(function (request, response)
{
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8080);
console.log('Server started');
```

- **`# apk -U add curl`**
- `# curl web.engr.oregonstate.edu/~brewsteb/CS312/server.js > /srv/server.js`

  OR
- **`# curl -L goo.gl/rsoKuT > /srv/server.js`**
- **`# cat /srv/server.js`**

Building a Webserver in a Container - Launch

SAY	**DEMONSTRATE**
• Start the server in the background	• `# node /srv/server.js &`
• See it	• `# jobs`
• Install curl if you haven't yet	• `# apk -U add curl`
• Test out the server: we're inside the container, so we can use localhost to target ourselves	• `# curl localhost:8080`
• Exit the container	• `# exit`
• **Huge caveat:** we'll need to start *that* specific, non-distributable container to get back to what we've done, as these changes cannot be made to the image (which is always read-only) in this manner	• `$ docker start -i webtest` • `# ls` • `# exit`
• Note that "restart" doesn't allow `-i`, and start doesn't need `-t`	

Building Our Own Image

- Recall that images are:
  - Read-only templates
  - What are *actually* distributed, since containers are only locally instanced
  - As large as the files contained inside them, obviously, so keep them small!
- If the containers need to do something in particular, and not simply provide a long-running service, then we can build a configuration into the image, that runs commands in any containerized instance of that image - this "bakes in" to the image all the work we'd done before
- Images are very difficult, if not impossible, to modify after they have been "built", so we write source code, then **build**, to produce images

Image Construction Theory

- Technically, images can be built from containers with "docker commit", but this is backwards thinking: do not do it!
- Images should be built with reproducible, transmittable plans, i.e. from source code, which is imported by a script we'll talk about next
- These kinds of source-to-image (S2I) methods allow us to track changes to our code that builds the image, with these changes being storable in version control systems like git

Dockerfiles

- We can build images that contain all those previous instructions, to be run automatically when a container is started, using **Dockerfiles**
- This allows us to apply the changes we had previously made to a container to the base image instead; we can then distribute that image

- A Dockerfile is thus a setup script used to *build a specific image*

- Distributing a Dockerfile *itself* is not the same as actually distributing a compiled binary image, which are the types of images we've been pulling and using, but it is a very bandwidth-efficient way to go, as Dockerfiles are simply small text files

Our Dockerfile

```
# Simple NodeJS Hello World server
FROM alpine:latest
MAINTAINER Benjamin Brewster <brewsteb@oregonstate.edu>

# During Build: Install the NodeJS runtime by running this command in the image
RUN apk -U add nodejs

# During build: Copy the server JS file into the image, store it at this location
COPY server.js /srv/server.js

# Expose the port 8080 for HTTP
EXPOSE 8080

# Run this command by default when containers spawned from this image start
CMD node /srv/server.js
```

Running Commands in an Image - Webserver

Say

- Make a directory to hold the relevant files
- Get into that directory
- Get the Dockerfile from Ben's website
- Get the NodeJS JS server file from Ben's website
- Build the image from our Dockerfile!
  - -t flag allows us to add our own tag
  - The current directory (.) is where we are doing the build, so all the relevant files (Dockerfile & server.js) need to be there
- Find the new image on our computer!

Demonstrate

- `$ mkdir NJSweb`
- `$ cd NJSweb`
- `$ curl web.engr.oregonstate.edu/~brewsteb/CS312/Dockerfile.NJSweb > Dockerfile`
- `$ curl web.engr.oregonstate.edu/~brewsteb/CS312/server.js > server.js`
- `$ docker build -t stonesand/brewpriv:NJSweb .`
- `$ docker images`

Running Commands in an Image - Webserver

Say

- Run the built image
  - −`rm`: Removes the container after it terminates
  - -d: Run the container detached, like a daemon, so we get our shell back (if you forget this, you might have to reset your VM!)
  - -p: Maps the 8080 port in use by the NodeJS server out to the Host, so that it can be accessed
- Test the NodeJS server, that we made, and distributed via image!
- Might as well push it to keep it

Demonstrate

- `$ docker run --rm -d -p 8080:8080 stonesand/brewpriv:NJSweb`
- `$ curl localhost:8080`
- `$ docker login --username stonesand --password-stdin < ~/.docker/creds`
- `$ docker push stonesand/brewpriv:NJSweb`

Let's write our own custom Docker Image

Say	Demonstrate
Make a directory for this new custom image	`$ mkdir hw`
Change into that dir	`$ cd hw`
Get the Dockerfile for our image	`$ curl web.engr.oregonstate.edu/~brewsteb/CS312/Dockerfile.hw > Dockerfile`
Discuss what's in the file	`$ cat Dockerfile`
Get the .c file we're going to build as part of the image	`$ curl web.engr.oregonstate.edu/~brewsteb/CS312/hw.c > hw.c`
Examine the program	`$ cat hw.c`
Get the script we're going to run when the container executes	`$ curl web.engr.oregonstate.edu/~brewsteb/CS312/compileAndRun.hw.sh > compileAndRun.hw.sh`
Check out the contents of the script	`$ cat compileAndRun.hw.sh`

Let's write our own custom Docker Image

Say

- Build the image
- Run the image - see the compile and run!
- Look at all this garbage!
- We can filter it all out using the same filtering switch (`-f`) that we used on containers, before
    - This command will complain about the containers based on these images still being in use, so use this:
- Much cleaner
- Docker might be full of containers
- Clean them up in a targeted manner

Demonstrate

- `$ docker build -t stonesand/brewpriv:hw .`
- `$ docker run stonesand/brewpriv:hw`
- `$ docker images`
- `$ docker image rm $(docker images -q -f dangling=true)`
- `$ docker image rm --force $(docker images -q -f dangling=true)`
- `$ docker images`
- `$ docker ps -a`
- `$ docker rm $(docker ps -aq -f exited=127)`

Docker in Production

- Docker containers can be organized into **Stacks** that provide a certain **Service**. These services could, for example, maintain X containers at moderate load, adding and subtracting more as needed, with 1 extra simply available for surges.
- As load increases, additional containers are spun up on other hardware, which hardware is altogether called the **Swarm**, keeping the load spread out.
- A **Swarm Manager** (a server) manages all of this
- This is all set up using a set of YAML files and a program called Docker **Compose**. This is covered in the Lab for this week.

Conclusion & Time for Questions

- Docker can be complicated, but once you get used to using it, you'll want to start creating images for everything, because:
- Once you understand how easy it is to tear down and stand back up a service with Docker, you'll never want to install something on bare metal again!
- Backing up Docker worker containers is trivial, since they are simply instantiated from an Image that should be stored locally and in the cloud
- DBs targeted by the worker containers still need traditional backup, but the DB services are just as easy to tear down and stand up