Cloud Basics for developers, Docker containers

CS634
Class 19

What we have learned so far

- It's pretty easy to set up a VM in the cloud, esp. if you've done it once before.
- It's not expensive, maybe $20/mo. for a single CPU.
- It's available from anywhere on the Internet.
- We are charged only for actual use, can bring the VM up and down easily to avoid charges. That's "on demand availability" and "measured service"
- We see it's possible to add computing power and disk resources by clicking a few buttons—that's "elasticity" and "self-service".
- We don't have to worry about fixing the system—though should do snapshots/backup to cover human-error scenarios

What we have learned so far, continued

Google Compute Engine:

- We can get an external IP address for the VM, so it can work as a server.
- Real user authentication is via Google sign-in, not system password.
- Multiple users can login to the VM, but each is a privileged user (i.e., on the development team)
- Ordinary users are handled via services exposed on the network: web server, web services, databases
- We can open up ports in the default firewall for our services (port 80 is open by default)

Compute Engine: what kind of cloud service is it?

Cloud service offerings are classified as follows

- Infrastructure as a service IaaS:
  - Just virtual machine supplied, you have to install OS
- Platform as a service PaaS:
  - VM with OS, basic tools, supplied ← Compute Engine VM
- Software as a service SaaS:
  - VM with OS, useful hosted apps, supplied: email, etc.
- Information as a service IaaS:
  - Specialized information available by web service, etc.
- Business process as a service BPaaS:
  - Multistep actions handled by service, example: PriceLine
- Functions as a service FaaS: AWS Lambda

Google App Engine: "Google's PaaS"

- Although Compute Engine is clearly a PaaS, it's not Google's official one
- The App Engine is a "managed PaaS"
- It uses Compute Engine VMs, but largely transparently.
- You create a web app or web services locally in your favorite language and upload it all with Google's gcloud tool
- You can see its status, disable/reenable services, modify parameters using web pages much like Compute Engine VMs.
- In some cases, you could login to one of the VMs for debugging.
- App Engine "Flexible" environment can accept software in containers

What cloud technologies do you need to know?

Useful summary: 9 Cloud Technologies You Need to Know for 2018

From its intro:

- Cloud revenues for the first half of 2017 totaled $63.2 billion. Compare to IBM revenue of $79 billion.
- The platform as a service (PaaS) segment of the market, saw particularly strong growth, with revenues increasing 50.2 percent year-over-year.
- Split into providers: for IaaS/Paas. Amazon on top, Google not, but growing fast (data as of Dec., 2017),shows total of $42 billion for IaaS/Paas.

The list of nine hot cloud technologies

1. Hybrid clouds: using private and public, for example
2. Containers: we'll study this
3. Kubernetes: "orchestration" of containers in production use
4. Serverless computing, or FaaS: Amazon Lambda
5. APIs, in sense of web services (covered in cs637)
6. Microservices: split "monolithic" apps into many pieces (which can live in containers) cs636 coverage, with linked videos.
7. Machine Learning: cs671 intro slides
8. Automation via DevOps (containers are important here)
9. Blockchain (cs646 is studying Berkeley blockchain workshops)

Four of these are related to containers!

Containers

Containers create a sandbox environment for a program to run in, isolating it from other programs and even the filesystem of the system it's running in, and its network.

- It does use the OS kernel, originally only Linux.
- Needs to provide its own filesystem, since isolated from the shared one.
- Needs to have its own network, since isolated from the shared one.
- Usually a single process runs inside the container, but more are allowed.
- Note that an ordinary process isolates memory from other processes, but shares the filesystem, and network ports.

If it's so isolated, how can it be useful at all?

- It can "expose" a network port, and this can be connected to, or mapped to a system port.
- Most communication to/from working containers is via TCP stream connections.
- For debugging or interactive use, can read/write from/to the terminal (almost the only thing that's still shared!)
- So can think of a container as a little machine with some hoses hanging out for connections to other such machines or the web. Also has a terminal connection.
- Database clients connect to the database server with a TCP connection, so ready to play in this environment.
- Example: mysql in a container, its client in another container, connected by the TCP connection, and the client also connected to the web.

Uses for containers

- Problem: program works fine on developer's system, fails or misbehaves on production system
- It's usually because some dependency is different: version of programming language, library, environment variables, …
- If developer puts it in a container, it carries with it all these dependencies, should run fine on production system.
- This idea is part of DevOps, movement to automate handling of software products, esp. delivery of software onto production systems.
- Similarly, the program should run fine in a cloud PaaS, or using Kubernetes.
- "Build once, run anywhere"
- Much smaller than VMs, the competition in a sense

If this is such a great idea, why did it take until 2013 to come about?

- Actually, some people were thinking about it, but Linux couldn't do the basic system support for it.
- Several needed Linux enhancements culminated in v 3.8 with capabilities for user-level sandboxing. Released Feb., 2013.
- 2013: Docker containers developed
- 2014: Docker 1.0 released, open-source
- 2014: Kubernetes released (helps with the production execution of many often-unrelated containers), open-source
- 2014: Compose 1.0 released (helps set up multiple communicating containers for a single app), open-source
- 2016: Docker for Windows Server 2016 and Windows 10 released (Microsoft responded quickly to this challenge!)

Docker Containers and Images

- Container: the executable object, like an executable file but holding a whole filesystem inside ready for the program.
- Docker Image: stored software in a format ready for use in a container. A container is built from one or more images. An image is something like a .class file, a template for building executables, not an executable itself.
  - Pre-built images are available from the Docker hub and elsewhere
  - You can build an image from your own software
  - Once you have an image, you can "run" it, passing various arguments. This will build and execute the container.
  - Usually containers are executed only once. They are considered disposable. The exited container can be examined.
  - Once installed on a host system, Docker provides the a docker command, and a docker daemon (dockerd) to live on the host system and carry out the docker commands.

Docker Hello World: last step of install

```
$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be
working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from
the Docker Hub. (amd64)
 3. The Docker daemon created a new container from that
image which runs the executable that produces the output
you are currently reading.
 4. The Docker daemon streamed that output to the Docker
client, which sent it to your terminal.
…
```

To redo this hello-world from scratch…

```
# remove image and any containers for it, re-run
eoneil@eonvm:~$ sudo docker rmi -f hello-world
Untagged: hello-world:latest
Untagged: hello-
world@sha256:97ce6fa4b6cdc079290b74cfebd...b8c38e979330d547d22ce1
Deleted:
sha256:f2a91732366c0332ccd7af2b9af81f549370f7a19acd460f87686bc7
eoneil@eonvm:~$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9bb5a5d4561a: Pull complete
Digest:
sha256:f5233545e43561214ca4891e1c3c563316ed8e237750d59bde73361e77
Status: Downloaded newer image for hello-world:latest
Hello from Docker!
This message shows that your installation appears to be working
correctly…
```

Hello world: it works, but where's the code?

- Can get some info about it from Docker Hub

(Google "docker hub hello-world" for ex.)

- Refs code in github
- Turns out it's in C, so not of much interest to us.

- We would like a hello-world written in Java
- Here is one tutorial that does it.

Using an Alpine container

- After the C hello-world, the tutorial does

```
sudo docker run alpine:latest "echo" "Hello, World‶
```

- Which downloads "alpine" and it does the echo: what is this all about?
- Alpine is a stripped-down Linux image (no kernel, just filesystem) with just a few Linux commands supported, including echo, and this command runs echo with argument "Hello, World"
- We can try others like

```
sudo docker run alpine:latest ls
```

- and see all the top-level directories of Linux listed
- To see something specific to Alpine, try
- `sudo docker run alpine:latest cat /etc/motd`
- Other containers allow this kind of access too, but have an app installed in them

Finally, the Java Hello

- This tutorial creates HelloWorld.java, compiles it on the host system, then creates a container for the .class file execution.
- In the homework, you'll also compile it in the container.
- For this approach, you could install java on your VM and do the compile, or just compile it on pe07 and transfer the .class file to your VM.
- After all, Java .class files are completely portable.
- OK, assume we have HelloWorld.class on our VM in a directory with the Dockerfile shown on the next slide…

Executing HelloWorld.class in a container

- We need to build an image, so we create a Dockerfile to say how. The tutorial has:

```
FROM alpine:latest
ADD HelloWorld.class HelloWorld.class
RUN apk --update add openjdk8-jre
ENTRYPOINT ["java", "-Djava.security.egd=
file:/dev/./urandom", "HelloWorld"]
```

- The all-caps commands are Dockerfile commands
- FROM alpine:latest means start from the Alpine slimmed-down Linux image, a very well-known resource
- apk is the Alpine package manager, like apt-get for Debian
- RUN apk … runs this command in the Alpine Linux already established, to install the JRE on that OS in the container-to-be

Executing HelloWorld.class in a container

```
FROM alpine:latest
ADD HelloWorld.class HelloWorld.class
RUN apk --update add openjdk8-jre
ENTRYPOINT ["java", "-Djava.security.egd
=file:/dev/./urandom", "HelloWorld"]
```

- So this starts with Linux, installs the JRE (Java binaries)
- Just like on our VMs, we'd use apt-get to install the JRE
- ADD copies HelloWorld.class from the current host directory into the developing image (as a "layer"), at the root of its filesystem, the default working directory
- We could use COPY here instead (for example)
  - COPY HelloWorld.class /
- ENTRYPOINT specs the default command (what happens if the user does "docker run <this-image>"), here run java with needed flag

That was a classic way to do it: OS + installed JRE

- But we can start from a JDK or JRE *image*
- See OpenJDK at Docker Hub
- Their example Dockerfile:

```
FROM openjdk:7
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN javac Main.java
CMD ["java", "Main"]
```

This starts from the JDK image, copies the current directory to /usr/src/myapp inside the container, sets the working directory within the container to /usr/src/myapp, runs the compiler and sets the default command of the container-to-be, the java command. Of course this will end up with a much bigger image.

Running this image…see Debian OS

Start from Main.java and the Dockerfile of last slide in a directory

Build an image of tag openjdk for easy running by name:

```
$ sudo docker build -t openjdk .
sudo docker run --rm openjdk
Hello World
```

We can use Linux commands to query the container environment: Here we are overriding the CMD java … with cat /etc/motd.

```
$ sudo docker run --rm openjdk cat /etc/motd
The programs included with the Debian GNU/Linux system are
free software…
```

We can run a shell *inside* the container: needs –it to allow terminal input:

```
$ sudo docker run --rm -it openjdk bash
root@dcf4d09b1b8a:/usr/src/myapp# lsDockerfile Main.class
Main.java
root@dcf4d09b1b8a:/usr/src/myapp# exit
Exit
$
```

Now we have Java working, what app?

- As discussed in the intro, working containers usually exchange data via TCP connections.
- Problem 2 in hw6 gives a simple example of a web service implemented in Java
- We'll talk to it over a TCP connection: send request to localhost/ping, get "pong" back
- We'll look at it more a little later

Accessing the host filesystem from a container using docker run –v …

- Suppose we wanted to containerize an app that reads and input file and produces an output file.
- We can use COPY to copy-in the input file, but these Dockerfile commands are long gone when the container executes to produce the output file.
- We could use the network to scp it out.
- Or use –v on the docker run command line to map a host directory into the container's filesystem.
  - For example **`docker run -v filedir:/mnt myimage`** maps host directory filedir (subdirectory of the Dockerfile directory) to /mnt inside the container
  - Then the app in the container just writes output to /mnt/data.dat and it shows up in filedir/data.dat on the host.
- This is used in problem 1 of hw6 to give a containerized JDK access to a user's java program directory. That's also an input-output situation.
- Also in problem 3, where mysql needs a stable place to store its data: the internal filesystem disappears when the container exits.

Image layers

- The image is built following the instructions in the Dockerfile, in layers, one layer for each command there. For our second java example:

```
FROM openjdk:7
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN javac Main.java
CMD ["java", "Main"]
```

- We can see the layers with the Docker history command: the first layers come from the steps we see above, which were done locally
- Then we see the layers of the downloaded image "openjdk:7", i.e,, the various commands that its Dockerfile used to create it
- Get OS, get OS tools, get JDK, set it up properly

Output of `docker history openjdk`

```
IMAGE          CREATED         CREATED BY                                       SIZE
6fbc97991540   41 hours ago    /bin/sh -c #(nop)  CMD ["java" "Main"]           0B
ccc921acf457   41 hours ago    /bin/sh -c javac Main.java                       409B
f8a0681a23e3   41 hours ago    /bin/sh -c #(nop) WORKDIR /usr/src/myapp         0B
9447bff1e77f   41 hours ago    /bin/sh -c #(nop) COPY dir:83c3b6b77ad65898c…    201B
8da12bfcb376   2 weeks ago     /bin/sh -c set -ex;   if [ ! -d /usr/share/m…    263MB
<missing>      2 weeks ago     /bin/sh -c #(nop)  ENV JAVA_DEBIAN_VERSION=7…    0B
<missing>      2 weeks ago     /bin/sh -c #(nop)  ENV JAVA_VERSION=7u171        0B
<missing>      5 weeks ago     /bin/sh -c #(nop)  ENV JAVA_HOME=/docker-jav…    0B
<missing>      5 weeks ago     /bin/sh -c ln -svT "/usr/lib/jvm/java-7-open…    33B
<missing>      5 weeks ago     /bin/sh -c {   echo '#!/bin/sh';   echo 'set…    87B
<missing>      5 weeks ago     /bin/sh -c #(nop)  ENV LANG=C.UTF-8              0B
<missing>      5 weeks ago     /bin/sh -c ap-get update && apt-get install…     2.1MB
<missing>      5 weeks ago     /bin/sh -c apt-get update && apt-get install…    123MB
<missing>      5 weeks ago     /bin/sh -c set -ex;  if ! command -v gpg > /…    0B
<missing>      5 weeks ago     /bin/sh -c apt-get update && apt-get install…    44.6MB
<missing>      5 weeks ago     /bin/sh -c #(nop)  CMD ["bash"]                  0B
<missing>      5 weeks ago     /bin/sh -c #(nop) ADD file:bc844c4763367b5f0…    123MB
```

Mystery of “<missing>” image ids

- Unfortunate text: not an error!
- Locally-created layers have image ids
- Downloaded images just have overall image id: layers have been put together
- The locally-created layers have a physical presence on the local machine, can be seen in docker daemon workspace /var/lib/docker/overlay2

```
sudo ls -l /var/lib/docker/overlay2/ffd94…19fe2-init
drwxr-xr-x 4 root root 4096 Mar 23 17:56 diff
-rw-r--r-- 1 root root 26   Mar 23 17:56 link
-rw-r--r-- 1 root root 289  Mar 23 17:56 lower
drwx------ 3 root root 4096 Mar 23 17:56 work
sudo ls -l /var/lib/docker/overlay2/ffd94…19fe2-init/diff
drwxr-xr-x 4 root root 4096 Mar 23 17:56 dev
drwxr-xr-x 2 root root 4096 Mar 23 17:56 etc
```

This gives a clue as to what’s going on: a layer contains a diff of a filesystem. This layer records changes to /dev and /etc. Looking inside etc, we see hostname, hosts, etc.

How Layers Work

- Each layer has the changed files for that action. Example on last slide: /etc/hostname changed by action, so new file is in the diff
- The layers stored in the image are read-only in the container environment.
- When the container is created from the image, another writable layer is added.
- The layers are mounted in place at the root / using a recent Linux enhancement (union filesystem) allowing multiple mounts at a single point, with the one mount's files being accessible if not overridden by another higher-priority mount's files. See UnionFS at Wikipedia if interested in details.
- The whole thing can be accessed as a normal filesystem by the process(es) running in the container.

Layer Caching and fast docker builds

- As we have seen, the individual layers are saved on the host system.
- Using a fairly careful algorithm, a saved layer image can be used for a build step instead of building it from scratch, or re-downloading it from elsewhere.
- This speeds up builds and esp. rebuilds, supporting the notion of a disposable container.
- Layers are now id'd by checksum to avoid possibility of tampering with them.
- The fact that we see them in /var/lib/docker/overlay2 and not /var/lib/docker/aufs means we're using the latest version, the "overlay2 driver".

Using the Network

- In problem 2 of hw6, you will work with PingPong.java
- What it does: if you send a request to http:localhost:8080/ping, you get "pong" back on the same TCP connection.
- To make this work:
- Build image: PingPong.java, JDK, OS gets put in the image with one port exported, for the client to connect to.
- Run image, with port mapped to some OS port
- We can use curl as a client, or a browser running on another system if the port is open externally (e.g., port 80)
- Or we could containerize a client…

Details on ports

- Dockerfile for PingPong has EXPOSE 8080
- This means TCP port 8080 of the container OS is ready for use from outside the container if the docker run command says how.
- From PingPong tutorial: use –p (or –publish) with docker run for port, also –d to run as daemon:

```
sudo docker run -d -p 8080:8080
```

- This binds exported container port 8080 to host port 8080

- Send GET request to localhost:8080/ping using curl:

```
curl localhost:8080/ping
pong
```

Using host port 80: not blocked by firewall

Another way: bind it to host port 80

First kill old run:

```
sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
5a10ecb3a5ee pingpong "java PingPong" 8 minutes ago Up 8
minutes 0.0.0.0:80->80/tcp, 8080/tcp
compassionate_franklin
$ sudo docker kill 5a10ecb3a5ee
5a10ecb3a5ee
```

Now run with different host port

```
sudo docker run -d -p 8080:80
```

Using curl to port 80:

```
curl localhost:80/ping
pong
```

From another system: browse to external IP of VM, URL /ping, see pong

My system (if up): http://35.190.172.174/ping

(I had to stop apache to free up port 80: sudo apachectl –k stop)

Still another way: use container's network

- We can use the exposed port without binding it to a host port—here's how: use docker run with –expose, no –p:

```
sudo docker run -d --expose 8080 pingpong
```

- Then find its usable container id using docker ps (see 2 executions)

```
sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED …
e14bb9b7858e pingpong "java PingPong" 38 seconds ago… ←one we want
0e4959d600f0 pingpong "java PingPong" 38 minutes ago…
```

Use container inspect to find its network

```
sudo docker inspect e14bb9b7858e | grep IPAddrss
 "SecondaryIPAddresses": null,
            "IPAddress": "172.17.0.3",
                    "IPAddress": "172.17.0.3",
```

Contact port 8080 at that address

```
curl 172.17.0.3:8080/ping
pong
```

Mysql in a container

- Hw6 Problem 3 considers a containerized mysql server.
- The default mysql at Docker Hub is now version 8.0, incompatible with our VM's mysql, so please specify v. 5.7 by replacing mysql in the original run command by mysql/mysql-server:5.7. See hw6.html.
- We don't have the Dockerfile here because we're pulling the prebuilt image from the Docker Hub.
- But the general rule is to export the expected port, here 3306, and that value reported in the startup output
- Again we find the container id and use inspect to get the IP address of the container's system.

Using the mysql server container

- The server container EXPORTs port 3306, so we can access the server in it once we know its IP address from “docker inspect <containerid>”
- But actually logging in requires an account and good password, and the server starts out nearly empty with only a root account with host=localhost.
- There are several ways to proceed. See MySQL 5.7 doc, including more secure methods involving one-time passwords.
- Hw6 shows how to log into mysql inside the container and fix up the root login to work from remote hosts (including the docker host system), then save that work in a new image using docker commit
- But it’s really better to use a Dockerfile to set up an image, because it documents the process.

Dockerfile to initialize mysql

```
FROM mysql/mysql-server:5.7
COPY mysql-setup.sql /docker-entrypoint-initdb.d
```

Mysql-setup.sql: (could do much more)

```
CREATE USER 'root'@'%' IDENTIFIED BY 'mypassword';
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%';
```

Build image:

```
sudo docker build -t mysql57a .
```

Run it (no longer need MYSQL_ROOT_PASSWORD here):

```
sudo docker run -d mysql57a
```

Find its IP address using inspect and log in from host:

```
mysql -h 172.17.0.3 -u root -p
```

Docker Volumes for mysql data

- Current mysql Docker images set up Docker "volumes" for their data by default, themselves held in /var/lib/mysql/volumes
- Inspect of container shows this JSON for the volume:


```
"Mounts": [{
"Type": "volume",
"Name":"c530dcad86c92b43f…ef09cd2b6911f",
"Source":
/var/lib/docker/volumes/c530dcad86c92b43f…ef09cd2b6911f/_data",
"Destination": "/var/lib/mysql",
"Driver": "local", "Mode": "", "RW": true, "Propagation": ""
}]
```


The container name is important: if the original container exits, you can start a new one referencing the volume by this name, and continue using the data.

```
sudo docker run -d --mount source=
c530dcad86c92b43f…ef09cd2b6911f,target=/var/lib/mysql mysql57a
```

Docker volumes can eat up your diskspace!

- If you try out mysql containers too many times, you can run out of the 10GB we have on a VM!
- Check for overall diskspace: df command

```
$ df
Filesystem       1K-blocks    Used Available Use% Mounted on
udev               1885404       0   1885404   0% /dev
tmpfs               379304   26744    352560   8% /run
/dev/sda1         10188088 8911484    736036  93% /   ←a problem!
tmpfs              1896508       0   1896508   0% /dev/shm
tmpfs                 5120       0      5120   0% /run/lock
tmpfs              1896508       0   1896508   0% /sys/fs/cgroup
tmpfs               379300       0    379300   0% /run/user/1000
```

Check if it's Docker's data…

```
$cd /var/lib/docker
$ sudo du -s $(sudo ls)     du output is in KB: do by directory
20      builder
236     containerd
336     containers
9088    image
72      network
2719224 overlay2  ←images and containers 2.7GB
20      plugins
4       runtimes
4       swarm
4       tmp
4       trust
2330864 volumes  ← volumes for mysql containers 2.3GB
```

Cleaning up Docker files

- Remove all stopped containers: they're disposable after all

sudo docker container prune

- Remove all untagged images, once containers are gone:

sudo docker image prune

- Remove all unused volumes

sudo docker volume prune

After this: df shows 71% use, 6.8GB, down from 8.9GB, with only 0.2GB in /var/lib/docker/volumes.

Should a database server be containerized?

- We have seen the complication of ensuring that the database data is persistent
- We have not even tackled the maintenance issues, like backups
- Typically we have one server, many clients for it
- The clients are less problematic to containerize, assuming they use the database to persist their data (as they should)
- So for many cases, containerizing the clients and leaving the database server "normally" installed is a win.
  - All the database tools work as expected.
  - The error log is where it is expected, etc.

A Containerized database app

- Let's containerize JdbcCheckup.java (itself unedited):

```
FROM java:8
COPY . /
WORKDIR /
RUN javac JdbcCheckup.java
CMD ["java", "-classpath", "mysql-connector-java-5.1.43-bin.jar:.",
"JdbcCheckup"]
```

- Note how we put the driver jar on the command line for the java command—it was copied into the container in the COPY command.

sudo docker build -t check2 .

sudo docker run -it check2

See next slide for output from run using mysql server on VM (i.e., uncontainerized server)

Containerized JdbcCheckup run

sudo docker run -it check2

```
Please enter information to test connection to the database
Using Oracle (o), MySql (m) or HSQLDB (h)? m
user: eoneil
password: eoneil
use canned MySql connection string (y/n): y
host: 10.142.0.2
```

←Internal IP address of VM

```
port (often 3306): 3306
using connection string: jdbc:mysql://10.142.0.2:3306/eoneildb
Connecting to the database...connected.
Hello World!
Your JDBC installation is correct.
```

Note that localhost won't work here: the client and server are running on different networks

Also, this needs the host's mysql server to be enabled for connections from non-localhost clients by commenting out the line "bind-address=127.0.0.1" in 50-server.cnfg

Security considerations

- Giving a person privileges to run docker means giving them root privileges on the host system, at least in effect.
- Worry: malicious code inserted into container via original build or patched in.
  - Code inside the container is running with root privileges inside its sandbox, so can look at everything COPY'd in or mounted, or generated as it runs.
  - Code inside the container can "call home":
    - Run bash inside a container and try "ssh user@pe07.cs.umb.edu", "curl topcat.cs.umb.edu:80": they work fine.
    - Can set up a tunnel to port 3306 on pe07 to use mysql or JDBC (need to specify IPv4 with -4 flag)
  - We see that there's more control over what ports can be used to reach the container than what ports the container code can reach
- Need firewalls: article says you need a Linux networking expert, to protect the host and the containers themselves

Summary

- We have covered how to write simple Dockerfiles for containerizing apps, and using database server images (or not) and a simple JDBC app.

Next steps you might try:

- Arrange for the JDBC app container to use the containerized mysql. This is currently done by setting up a network among the containers (surprisingly easy).
- Have two JDBC apps using the same mysql this way.
- Have a whole cs636-style web app using mysql: can containerize tomcat with app in its filesystem, and separately mysql, or even Oracle.
- Try out [Docker Compose]to run multi-container apps.