1. **CHROOT**

* Make a new folder in your home directory via mkdir /my-new-root.
* Inside that new folder, run  echo "my super-secret thing" >> /my-new-root/secret.txt
* Now try to run chroot /my-new-root bash and see the error it gives you.
* You should see something about failing to run a shell or not being able to find bash. That's because bash is a program, and your new root wouldn't have bash to run.
* So, let's fix that! Run:

mkdir /my-new-root/bin

cp /bin/bash /bin/ls /my-new-root/bin/

chroot /my-new-root bash

* + - Still not working! The problem is that these commands rely on libraries to power them, and we didn't bring those with us. So, let's do that too. Run ldd /bin/bash. This prints out something like this:
* ldd /bin/bash
* These are the libraries we need for bash. Let's go ahead and copy those into our new environment.

mkdir /my-new-root/lib /my-new-root/lib64

* Then we need to copy all those paths (ignore the lines that don't have paths) into our directory. Make sure you get the right files in the right directory

cp /lib/x86\_64-linux-gnu/libtinfo.so.6 /lib/x86\_64-linux-gnu/libc.so.6 my-new-root/lib

cp /lib64/ld-linux-x86-64.so.2 my-new-root/lib64

* Do it again for ls. Run ldd /bin/ls

cp /lib/x86\_64-linux-gnu/libselinux.so.1 /lib/x86\_64-linux-gnu/libc.so.6 /lib/x86\_64-linux-gnu/libpcre2-8.so.0 my-new-root/lib

cp /lib64/ld-linux-x86-64.so.2 my-new-root/lib64/

* Note: For different linux distro, the packages will be different so don’t blindly copy the libraries from the document

## The problem with chroot alone

* chroot /my-new-root bash (I'll refer to the chroot'd environment as #1)
* Start another terminal session on the same Linux machine (I'll refer to the new session as #2)

## Run tail -f /my-new-root/secret.txt & in #2. This will start an infinitely running process in the background.

## Run ps to see the process list in #2 and see the tail process running. Copy the PID (process ID) for the tail process.

## In #1, the chroot'd shell, run kill <PID you just copied> .This will kill the tail process from inside the chroot'd environment. This is a problem because that means chroot isn't enough to isolate someone. We need more barriers. This is just one problem, processes, but it's illustrative that we need more isolation beyond just the file system.

1. **Namespaces**

* namespaces and cgroups are a bit more nebulous to understand but no less important. Both next two features are for security and resource management.
* Let's say you're running a big server that's in your home and you're selling space to other people (that you don't know) to run their code on your server. What sort of concerns would you have? Let's say you have Alice and Bob who are running e-commerce services dealing with lots of money. They themselves are good citizens of the servers and minding their own business. But then you have Eve join the server who has other intentions: she wants to steal money, source code, and whatever else she can get her hands on from your other tenants on the server. If just gave all three them root access to server, what's to stop Eve from taking everything?
* Your first line of defence is that you could log them into chroot'd environments and limit them to only those. Great! Now they can't see each other’s files. Problem solved? Well, no, not quite yet. Even though she can't see the files, she can still see all the processes going on the computer. She can kill processes, unmount filesystem and potentially even hijack processes.
* Enter namespaces. Namespaces allow you to hide processes from other processes. If we give each chroot'd environment different sets of namespaces, now Alice, Bob, and Eve can't see each other’s processes (they even get different process PIDs, or process IDs, so they can't guess what the others have) and you can't steal or hijack what you can't see!

**Safety with namespaces**

* Install debootstrap - debootstrap is a tool which will install a Debian/Ubuntu base system into a subdirectory of another, already installed system.
* Install debootstrap

apt-get update -y

apt-get install debootstrap -y

debootstrap --variant=minbase bionic /better-root

* Now head into the new namespace, chroot environment

unshare --mount --uts --ipc --net --pid --fork --user --map-root-user chroot /better-root bash

mount -t proc none /proc # process namespace

mount -t sysfs none /sys # filesystem namespace

mount -t tmpfs none /tmp # filesystem namespace

Now try our previous exercise again.

* Run tail -f /my-new-root/secret.txt & from main root
* Run ps aux , grab pid for tail
* Run kill <pid for tail> in the new root, see that it doesn't work

# **Cgroups**

* + we still have a problem. Every isolated environment has access to all physical resources of the server. There's no isolation of physical components from these environments.
  + Enter the hero of this story: cgroups, or control groups. Google saw this same problem when building their own infrastructure and wanted to protect runaway processes from taking down entire servers and made this idea of cgroups so you can say "this isolated environment only gets so much CPU, so much memory, etc. and once it's out of those it's out-of-luck, it won't get any more."
* outside of unshare'd environment get the tools we'll need here

apt-get install -y cgroup-tools htop

* create new cgroups

cgcreate -g cpu,memory,blkio,devices,freezer:/sandbox

* add our unshare'd env to our cgroup

ps aux # grab the bash PID that's right after the unshare one

cgclassify -g cpu,memory,blkio,devices,freezer:sandbox <PID>

* list tasks associated to the sandbox cpu group, we should see the above PID

cat /sys/fs/cgroup/cpu/sandbox/tasks

* show the cpu share of the sandbox cpu group, this is the number that determines priority between competing resources, higher is is higher priority

cat /sys/fs/cgroup/cpu/sandbox/cpu.shares

* Limit usage at 5% for a multi core system

cgset -r cpu.cfs\_period\_us=100000 -r cpu.cfs\_quota\_us=$[ 5000 \* $(getconf \_NPROCESSORS\_ONLN) ] sandbox

* Set a limit of 80M

cgset -r memory.limit\_in\_bytes=80M sandbox

* Get memory stats used by the cgroup

cgget -r memory.stat sandbox

* in terminal session #2, outside of the unshare'd env
* htop # will allow us to see resources being used with a nice visualizer
* in terminal session #1, inside unshared'd env

yes > /dev/null # this will instantly consume one core's worth of CPU power

* notice it's only taking 5% of the CPU, like we set
* in terminal session #1, inside unshare'd env

yes | tr \\n x | head -c 1048576000 | grep n # this will ramp up to consume ~1GB of RAM

* notice in htop it'll keep the memory closer to 80MB due to our cgroup
* as above, connect with a third terminal to see it work outside of a cgroup

So, while this is a container at its most basic sense, we haven't discussed more advance topics like networking, deploying, bundling, or anything else that something like Docker takes care of for us. But now you know at its most base level what a container is, what it does, and how you could do this yourself

# **Getting Set Up with Docker**

* So it's much easier to do what we did with Docker. Run this command:

docker run -it ubuntu:bionic

* This will drop you into an Alpine ash shell inside of a container as the root user of that container. When you're done, just run exit or hit CTRL+D.
* By default, containers run and then exit as soon as they're done.
* The -it or --interactive tag says you want to be dropped into the container interactively so you can run commands and inspect the container.
* docker run ubuntu:bionic ls

The ls part at the end is what you pass into the container to be run. As you can see here, it executes the command, outputs the results, and shuts down the container.

* So now what if we want to detach the container running from the foreground

docker run --detach -it ubuntu:bionic

or

docker run -dit ubuntu:bionic

* So, it prints a long hash out and then nothing. Well, it's running in the background. So how do we get a hold of it?

docker ps

This will print out all the running containers that Docker is managing for you. You should see your container there. So copy the ID or the name and say:

* docker attach <ID or name>

# e.g.: “docker attach 20919c49d6e5”

This allows you to attach a shell to a running container

* To kill this container without attaching to it. Run docker ps, get the IDs or names of the containers you want to kill and say:

docker kill <IDs or names of containers>

# e.g. `docker kill 20919c49d6e5`

* Let's make it a bit easier to keep track of these. Try this

docker run -dit --name my-ubuntu ubuntu:bionic

docker kill my-ubuntu

* Now you can refer to these by a name you set. But now if you tried it again, it'd say that my-ubuntu exists.
* If you run docker ps --all you'll see that the container exists even if it's been stopped. That's because Docker keeps this metadata around until you tell it to stop doing that
* You can run docker rm my-ubuntu which will free up that name or you can run docker container prune to free up all existing stopped containers
* In the future you can just do

docker run --rm -dit --name my-ubuntu ubuntu:bionic

docker kill my-ubuntu

* This will automatically clean up the container when it's done.

# **Docker CLI**

# Let's take a look at some more cool features of the Docker CLI.

* **pull**
* pull allows you to pre-fetch container to run

docker pull node:8

docker run -it node:8 bash

# notice it's already loaded and cached here; it doesn't redownload it

* **inspect**
  + docker inspect node
  + This will dump out a lot of info about the container. Helpful when figuring out what's going on with a container
* **pause / unpause**
* As it looks, these pauses or unpause all the processes in a container. Feel free to try

docker run -dit jturpin/hollywood hollywood

docker ps # see container running

docker pause <ID or name>

docker ps # see container paused

docker unpause <ID or name>

docker ps # see container running again

docker kill <ID or name> # see container is gone

* **Exec**
  + This allows you to execute a command against a running container. This is different from docker run because docker run will start a new container whereas docker exec runs the command in an already-running container.

docker run -dit jturpin/hollywood hollywood

docker ps # grab the name or ID

docker exec <ID or name> ps aux # see it output all the running processes of the container

* **History**
  + We'll get into layers in a bit but this allow you to see how this Docker image's layer composition has changed over time and how recently.

docker history node

* **Info**
  + Dumps a bunch of info about the host system. Useful if you're on a VM somewhere and not sure what the environment is.

docker info

* **Top**
  + Allows you to see processes running on a container (similar to what we did above)

docker run mongo

docker top <ID outputted by previous command>

* **Logs**
  + Very useful to see the output of one of your running containers.

docker run -d mongo

docker logs <id from previous command>

# See all the logs

* **Search**
  + If you want to see if a container exists on Docker Hub (or whatever registry you're connected to), this will allow you to look.

docker search python

# See all the various flavours of Python containers you can run

docker search node

# See all the various flavours of Node.js containers you can run

1. **Dockerfile**

* Docker has a special file called a Dockerfile which allows you to outline how a container will be built. Each line in a Docker file is a new a directive of how to change your Docker container.
* A big key with Docker container is that they're supposed to be disposable. You should be able to create them and throw them away as many times as necessary.
* Let's make the most basic Dockerfile ever. Let's make a new folder, maybe on your desktop. Put a file in there called Dockerfile (no extension.) In your file, put this.

FROM node:12-stretch

CMD ["node", "-e", "console.log(\"hi lol\")"]

* The first thing on each line (FROM and CMD in this case) are called instructions. They don't technically have to be all caps but it's convention to do so so that the file is easier to read.
* Each one of these instruction incrementally changes the container from the state it was in previously, adding what we call a layer.
* Let's go ahead and build our container. Run (from inside of the directory of where your Dockerfile is)

docker build .

* Grab the hash from your build and run

docker run <ID>

* It's a little inconvenient to always have to refer to it by ID, it'd be easier if it had a name. So let's do that! Try

docker build . --tag my-node-app

## or -t instead of --tag

docker run my-node-app

# **Build a Node.js App**

* So now let's dig into some more advance things you can do with a Dockerfile. Let's first make our project a real Node.js application. Make a file called **index.js** and put this in there.

const hapi = require("@hapi/hapi");

async function start() {

const server = hapi.server({

host: "0.0.0.0",

port: process.env.PORT || 3000

});

server.route({

method: "GET",

path: "/",

handler() {

return { success: true };

}

});

await server.register({

plugin: require("hapi-pino"),

options: {

prettyPrint: true

}

});

await server.start();

return server;

}

start().catch(err => {

console.log(err);

process.exit(1);

});

* Add a Dockerfile in the node project root directory

FROM node:12-stretch

COPY . .

RUN npm ci

CMD ["node", "index.js"]

* This will copy your index.js file from your file system into the Docker file system (the first index.js is the source and the second index.js is the destination of that file inside the container.)
* We then instruct the CMD to start the server when we finally do run the container. Now run

docker build -t my-node-app .

docker run --init my-node-app

* Now your Node.js app is running inside of a container managed by Docker
* But one problem, how do we access it? If you open locahlost:3000 now, it doesn't work!
* We have to tell Docker to expose the port. So let's do that now. Stop your container from running and run it again like this

docker run --init --publish 8000:3000 my-node-app

* The publish part allows you to forward a port out of a container to the host computer. In this case we're forwarding the port of 3000 (which is what the Node.js server was listening on) to port 8000 on the host machine.
* you'd open localhost:8000 to see the server.
* Right now, we're putting our app into the root directory of our container and running it as the root user. This both messy and unsafe.
* If there's an exploit for Node.js that get released, it means that whoever uses that exploit on our Node.js server will doing so as root which means they can do whatever they want.
* So, let's fix that. We'll put the directory inside our home directory under a different user.

FROM node:12-stretch

USER node

RUN mkdir /home/node/code

WORKDIR /home/node/code

COPY --chown=node:node . .

RUN npm ci

CMD ["node", "index.js"]

* You can again rebuild the docker image and run it

docker build -t my-node-app .

docker run --init my-node-app

# **Layers**

* Go make any change to your Node.js app. Now re-run your build process. Docker is smart enough to see the FROM, RUN, and WORKDIR instructions haven't changed and wouldn't change if you ran them again so it uses the same containers it cached from the previous, but it can see that your COPY is different since files changed between last time and this time, so it begins the build process there and re-runs all instructions after that.

FROM node:12-stretch

USER node

RUN mkdir /home/node/code

WORKDIR /home/node/code

COPY --chown=node:node package\*.json ./

RUN npm ci

COPY --chown=node:node . .

CMD ["node", "index.js"]

# **Bind Mounts**

* Bind mounts allow you to mount files from your host computer into your container. This allows you to use the containers a much more flexible way than previously possible
* Let's use the “my-node-app” container to serve directly from it and run this command in the node project root directory on your machine.
* docker run --mount type=bind,source="$(pwd)",target=/home/node/code -p 8000:3000 –-init my-node-app
* This is how you do bind mounts. It's a bit verbose but necessary. Let's dissect it.
* We use the --mount flag to identify we're going to be mounting something in from the host.
* As far as I know the only two types are bind and volume. Here we're using bind because we to mount in some piece of already existing data from the host.
* In the source, we identify what part of the host we want to make readable-and-writable to the container. It must be an absolute path which is why use the "$(pwd)" to get the present working directory to make it an absolute path.
* The target is where we want those files to be mounted in the container.

# **Volumes**

* Bind mounts are great for when you need to share data between your host and your container as we just learned. Volumes, on the other hand, are so that your containers can maintain state between runs.
* So, if you have a container that runs and the next time it runs it needs the results from the previous time it ran, volumes are going to be helpful
* Volumes can not only be shared by the same container-type between runs but also between different containers.
* Let's make a quick Node.js app that reads from a file that a number in it, prints it, writes it to a volume, and finishes. Create a new Node.js project.

mkdir docker-volume

cd docker-volume

touch index.js Dockerfile

* Inside that Node.js file, put this:

const fs = require("fs").promises;

const path = require("path");

const dataPath = path.join(process.env.DATA\_PATH || "./data.txt");

fs.readFile(dataPath).then(buffer => {

const data = buffer.toString();

console.log(data);

writeTo(+data + 1);

}).catch(e => {

console.log("file not found, writing '0' to a new file");

writeTo(0);

});

const writeTo = data => {

fs.writeFile(dataPath, data.toString()).catch(console.error);

};

* If it just run it right now, it'll create a data.txt file with 0 in it. If you run it again, it'll have 1 in there and so on. So, let's make this work with volumes.
* In the Dockerfile put this code

FROM node:12-alpine

COPY --chown=node:node . /src

WORKDIR /src

CMD ["node", "index.js"]

* Now run

docker build --tag=incrementor .

docker run incrementor

* Every time you run this it'll be the same thing. This is nothing is persisted once the container finishes. We need something that can live between runs.
* So, without having to rebuild your container, try this

docker run --env DATA\_PATH=/data/num.txt --mount type=volume,src=incrementor-data,target=/data incrementor

# **Networking with Docker**

* There are several ways of doing networking within Docker and all of them work differently depending which operating system you're on.
* We're going to deal with the simplest, the bridge networks. There is a default bridge network running all the time.
* If you want to check this out, run docker network ls. You'll see something like this:

$ docker network ls

NETWORK ID NAME DRIVER SCOPE

xxxxxxxxxxxx bridge bridge local

xxxxxxxxxxxx host host local

xxxxxxxxxxxx none null local

* The bridge network is the one that exists all the time and we could attach to it if we want to, but again Docker recommends against it so we'll create our own. There's also the host network which is the host computer itself's network. The last network with the null driver is one that you'd use if you wanted to use some other provider or if you wanted to do it manually yourself.
* Go ahead and run docker network create --driver=bridge app-net
* Once you've done that, let's start a MongoDB server. Run docker run -d --network=app-net -p 27017:27017 --name=db --rm mongo:3
* The --rm means toss all that stuff as soon as the container finishes and free up that name again.
* If we didn't use --rm , we'd have to run docker rm db before restarting our db container since when it stops a container, it doesn't delete it and its logs and meta data until you tell it to.
* Create a new folder for a nodejs project

npm init -y

npm install @hapi/hapi [mongodb@3.3](mailto:mongodb@3.3) dotenv

* Create Index.js file and put this:

const hapi = require("@hapi/hapi");

const { MongoClient } = require("mongodb");

require('dotenv').config();

const url = process.env.MONGO\_CONNECTION\_STRING || "mongodb://localhost:27017";

const dbName = "dockerApp";

const collectionName = "count";

async function start() {

const client = await MongoClient.connect(url);

const db = client.db(dbName);

const collection = db.collection(collectionName);

const server = hapi.server({

host: "0.0.0.0",

port: process.env.PORT || 3000

});

server.route({

method: "GET",

path: "/",

async handler() {

const count = await collection.count();

return { success: true, count };

}

});

server.route({

method: "GET",

path: "/add",

async handler() {

const res = await collection.insertOne({});

return { inserted: res.insertedCount };

}

});

await server.start();

return server;

}

start().catch(err => {

console.log(err);

process.exit(1);

});

* Use this Dockerfile

FROM node:12-stretch

USER node

RUN mkdir /home/node/code

WORKDIR /home/node/code

COPY --chown=node:node package-lock.json package.json ./

RUN npm ci

COPY --chown=node:node . .

CMD ["node", "index.js"]

* So build the container and run it using the following commands:

docker build --tag=my-app-with-mongo .

docker run -p 3000:3000 --network=app-net --env MONGO\_CONNECTION\_STRING=mongodb://db:27017 my-app-with-mongo

* Now open localhost:3000 and you can add the count by triggering localhost:3000/add

1. **Docker-compose**

* This may be one of the most useful features you learn about Docker. We've been mixing various different facets of deploying your app to production and creating development environments. This feature in particular is geared much more for development environments.
* you need to coordinate multiple containers when you're doing local dev and you've seen in the previous chapter, networking, that it's possible if a bit annoying.
* With Docker Compose we simplify this a lot. Docker Compose allows us the ability to coordinate multiple containers and do so with one YAML file.
* Okay so let's get our previous app working: the one with a MongoDB database being connected to by a Node.js app. Create a new file in the root directory of your project called docker-compose.yml and put this in there:

version: "1"

services:

web:

build: .

ports:

- "3000:3000"

volumes:

- .:/home/node/code

- /home/node/code/node\_modules

links:

- mydb

environment:

MONGO\_CONNECTION\_STRING: mongodb://db:27017

mydb:

image: mongo:3

* Remember to turn off any container exposed to the port you want to host your application or database
* In service we define the containers we need for this particular app. We have two: the web container (which is our app) and the db container which is MongoDB.
* We then identify where the Dockerfile is with build, which ports to expose in ports, which volumes to make in volumes (here we're mounting in our code so that we can keep code without having to rebuild the image), and the environment variables using that field.
* run docker-compose build
* The one interesting one here is the links field. In this one we're saying that the web container needs to be connected to the db container
* This means Docker will start this container first and then network it to the web container.
* The db container is pretty simple: it's just the mongo container from Docker Hub. This is actually smart enough to expose 27017 as the port and to make a volume to keep the data around between restarts so we don't actually have to do anything for that.
* just run docker-compose up