Northeastern University

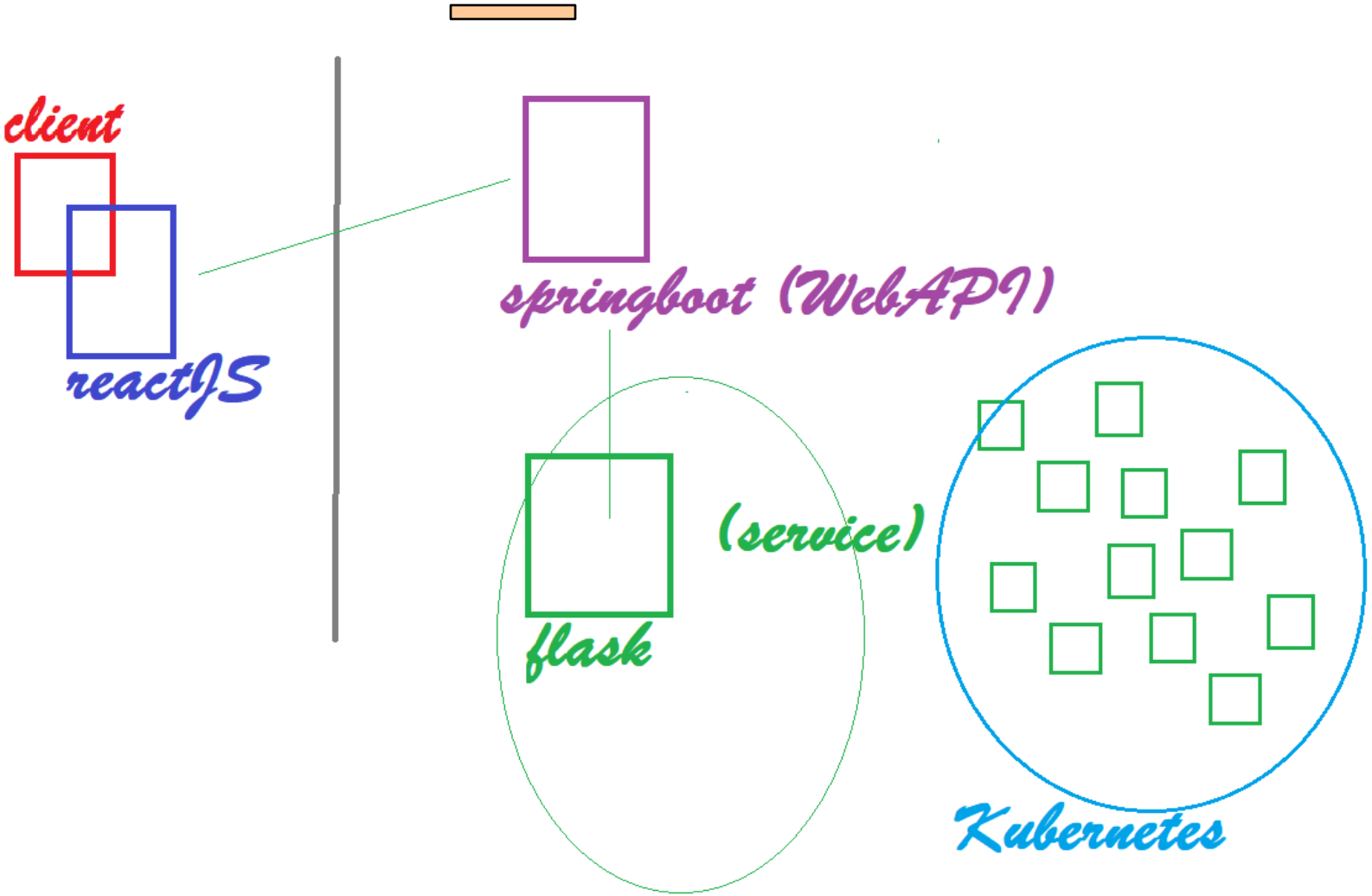
CSYE 7220 DevOps
Lecture 4 Containerizing apps

23 May 2024

The 3 commandments for enterprise Web apps

- The only communication channel between a server and its clients is through a Web App/WebAPI
  - There are 2 kinds of Web functionalities:
    - *Web app*: Web page (HTML + CSS + JS)
    - *WebAPI*: downloads static data (text/images/videos) usually through a RESTful API
      - REST is a stateless exchange through 4 HTTP verbs: GET/PUT/POST/DELETE
- You shall use both a client-side as well as a server-side stack
  - A client-side stack like `React` is essentially a local Web App
  - Why do we need a server-side Web app, then?
- How to design a distributed system:
  - s2s comm channel is a WebAPI, and c2s channel is a WebApp (be or fe)
  - Add a client-side stack for performance (alleviate bottleneck on the server)
  - On the server-side, you are free to have 1 million endpoints communicate with another million endpoints

µservice app arch diagram



Part 1

INSTALLS

You have some options..

- So, read first!

Install Docker Desktop (DD)

- Docker Desktop is *proprietary* 😱
- https://docs.docker.com/desktop/install/mac-install/
  - Using Intel Chip
  - Using M1/M2 Chip
- https://docs.docker.com/desktop/install/windows-install/
  - Using WSL 2 backend
  - Using Hyper-V backend and Windows containers

Or, install Docker Engine (Docker CE)

- **Docker Engine, a.k.a. Docker Community Edition (Docker CE), is Open Source**😻
- **https://docs.docker.com/engine/install/**
- Docker Desktop uses Docker Engine as one of its components
- Docker Engine is an open-source containerization technology for building and containerizing apps
- Docker Engine acts as a client-server application with:
  - A server with a long-running daemon process dockerd.
  - APIs which specify interfaces that programs can use to talk to and instruct the Docker daemon.
  - A command line interface (CLI) client docker

What's the difference?

- Docker Desktop is a VM + GUI with some extra features:
  - The new extensions
  - A single-node Kubernetes "cluster"
- Inside the virtual machine there is Docker CE (Docker Community Edition) daemon

 's in the details

- Docker containers run inside a VM
- The Docker network exists ***only inside the VM***, not on your host!
- This is also true for local volumes
- Even if you use the host network mode (`docker run --net host`) it will be the host network *of the VM*, not your actual physical host
- This is important when you try to access container IP addresses directly
- Docker containers run inside the VM but Docker Desktop hides that from you so you don't need to set up a VM and the associated client-server connection. Docker gives you the same experience with the same Linux kernel on any platform
- Having a VM is required on Windows and MacOS if you want to run Linux containers, because they use a Linux kernel
- Only on a Linux OS is a VM *not necessary*

Wait, why do I need a VM on the Mac?

- **Isn't a MAC actually a Linux OS?**
- **No! MacOS is based on UNIX BSD, not Linux. Although at the command line they look the same, they are not. They look the same just because they are based on the same POSIX standard**
- **So Docker for Mac runs in a *LinuxKit VM* and recently switched to the *Virtualization Framework* instead of *HyperKit***

The story of Mac

- What we call macOS today is based on NeXTstep, the operating system developed by NeXT Computing in the 1980s
- NeXT was the company that Steve Jobs founded after he left Apple in 1985
- He started a new company and hired a man named Avie Tevanian as his chief of software development. Tevanian had been one of the programmers who had developed the BSD Mach kernel at Carnegie Mellon University, and Jobs asked him to create a new multitasking OS based on it
- Apple decided to buy one from somebody who could get it right. After flirting with Microsoft and BeOS, they decided to buy NeXT out and acquire its technology, which had the side effect of bringing Jobs back into Apple
- Tevanian, who became Apple's new head of software development, then reworked NeXTstep into Mac OS X
- The modern Mac operating system is therefore based on a kernel from BSD, with a ton of Apple proprietary stuff running on top of it

The story of Mac

- You can actually download the source code for the kernel and other open-source components of macOS (it's called Darwin), but it's so different from the full macOS that it can't even run Mac software
- Linux was developed totally independently. The GNU user and tools have been in development since as early as 1983, but the Linux kernel wasn't written until 1991, with the first GNU/Linux distros following in 1992
- NeXTstep, by contrast, was released on 18 September 1989, and was already a polished OS ready to be used for serious work. Nothing based on Linux got to that point for several more years

What does Dino do?

- He uses *Oracle VirtualBox*, a hosted (Type II) hypervisor for x86 virtualization owned by Oracle Corporation
- VirtualBox was originally created by *InnoTek Systemberatung GmbH*, which was then acquired by *Sun Microsystems* in 2008, which was then acquired by *Oracle* in 2010
- He installs *Docker Toolbox,* which contains Docker CE and Oracle VirtualBox
- And he uses Docker CE instead of Docker Desktop
- Because Dino ❤️ Open Source
- But that requires Dino to start and control the VM, whereas using DD, you do not have to do that
  - My VM runs at ***http://192.168.99.100***
- So, DD is easier, requires less work, and CE is more complex
- CE is more transparent but requires more work
- You choose what is good for you ☺

Docker Toolbox

- Docker Toolbox is officially deprecated because Docker the Company wants you to use DD, of course 😒
- So it's tough to google for!
- Docker Toolbox includes the following Docker tools:
  - Docker CE
  - Docker CLI client for running Docker Engine to create images and containers
  - Docker Machine so you can run Docker Engine commands from Windows terminals
  - Docker Compose for running the docker-compose command
  - Kitematic, the Docker GUI
  - the Docker QuickStart shell preconfigured for a Docker command-line environment
  - *Oracle VM VirtualBox* 😁

Docker Toolbox on Windows

- Because Docker Engine daemon uses Linux-specific kernel features, you can't run Docker Engine natively on Window
  - Instead, you must use the Docker Machine command, `docker-machine`, to create and attach to a small Linux VM on your machine. This VM hosts Docker Engine for you on your Windows system
- Make sure your Windows system supports Hardware Virtualization Technology and that virtualization is enabled
- If not, go buy new laptop🤪
- https://github.com/docker-archive/toolbox/blob/master/docs/toolbox_install_windows.md

Docker Toolbox on MAC

- https://github.com/docker-archive/toolbox
- I've never ran Docker toolbox from the Mac, so maybe easier to run DD on Mac
- Probably easier to run DD on Windows, too

You can also

- Just install Docker CE and Oracle VirtualBox
- That’s all we’re going to use, essentially

The goodies of DD

- https://www.docker.com/blog/the-magic-behind-the-scenes-of-docker-desktop/
- Docker Desktop is way easier to use
- But you're selling your soul to…

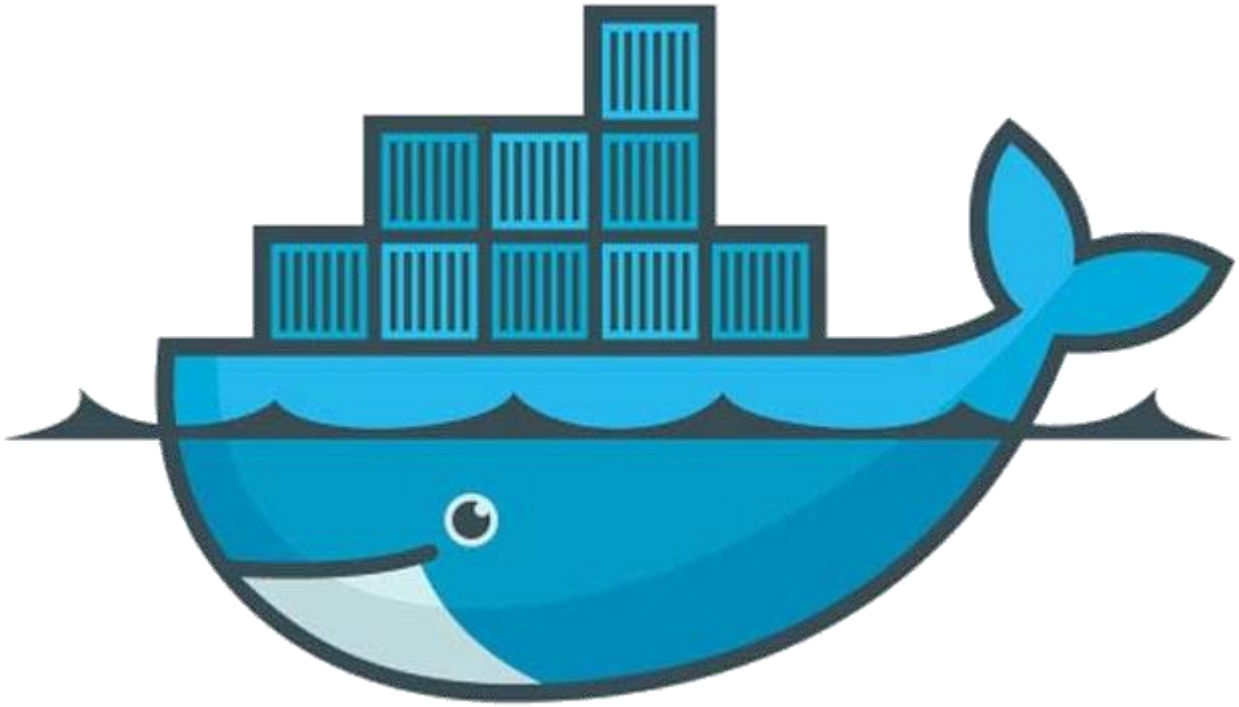
- **Now, decide…**

Part 2

VIRTUALIZATION

Docker

- Docker is the preeminent tool for containerization
- So we need to learn it ***well***

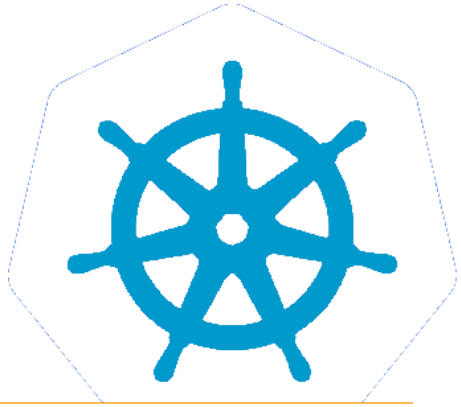


Application Server –Next Generation

- ***How to manage containers on the Cloud?***
- ***Kubernetes*** is quickly becoming the new standard for deploying and managing software in the cloud
  - A new kind of application server based on Container virtualization
- Official documentation
  - The various parts of the *Kubernetes Control Plane*, such as the Kubernetes Master and *kubelet processes*, govern how Kubernetes communicates with your *cluster*. The Control Plane maintains a record of all of the *Kubernetes Objects* in the system and runs continuous control loops to manage those objects' *state*. At any given time, the Control Plane's *control loops* will respond to changes in the cluster and work to make the actual state of all the objects in the system match the desired state that you requested
  - **https://kubernetes.io/docs/concepts/**

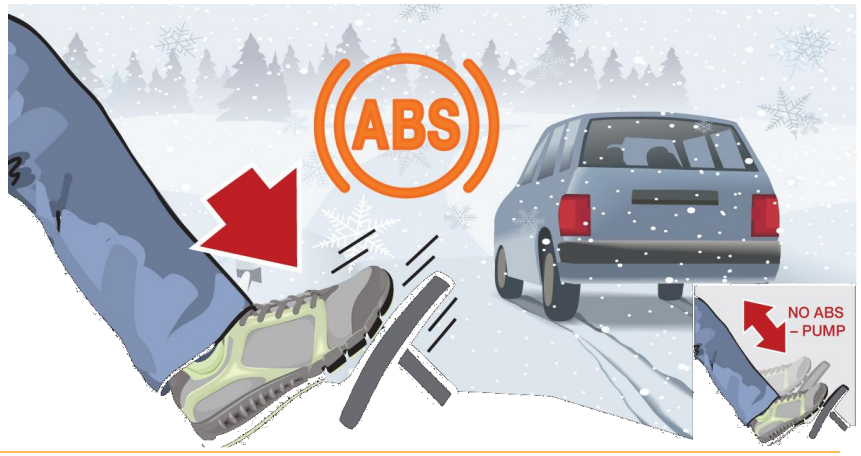
Why relevant to DevOps

- **Even though we van deploy software running *natively* on VMs on the Cloud, most software is deployed in *containers* instead of VMs, in order to fully maximize VM usage, simplify complexity, and leverage open source to manage it**
- **Moreover, logical containers and physical containers can be further virtualized so that logical containers are free to "move around" in order to fully maximize container usage and provide seamless scalability**
  - **Logical containers are called *microservices***
- **Google did this with their own internal operations, and then decided to open source it: `Kubernetes`**
- Kubernetes ("koo-burr-NET-eez") is the conventional pronunciation of a Greek word, κυβερνήτης, meaning "helmsman" or "pilot
  - Helps you navigate containerized applications on public (or private) Clouds



What is Virtualization?

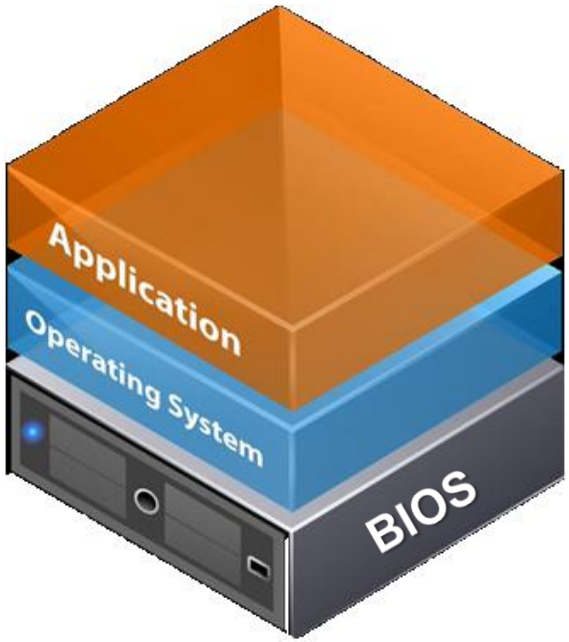
- Virtualization is the ***abstraction*** of computer resources
  - Abstraction: separating layers normally coupled
    - Anti-lock brakes: brake pedal connected to brakes through an intermediate layer
    - Town utilities: water is pumped in one place and then distributed
- Abstraction enables ***flexibility***
  - *Hardware*: multiple operating systems run on one computer
  - *Software*: programs with conflicting requirements may run concurrently
  - *Operating System*: distinct user or server environments may coexist inside one running operating system

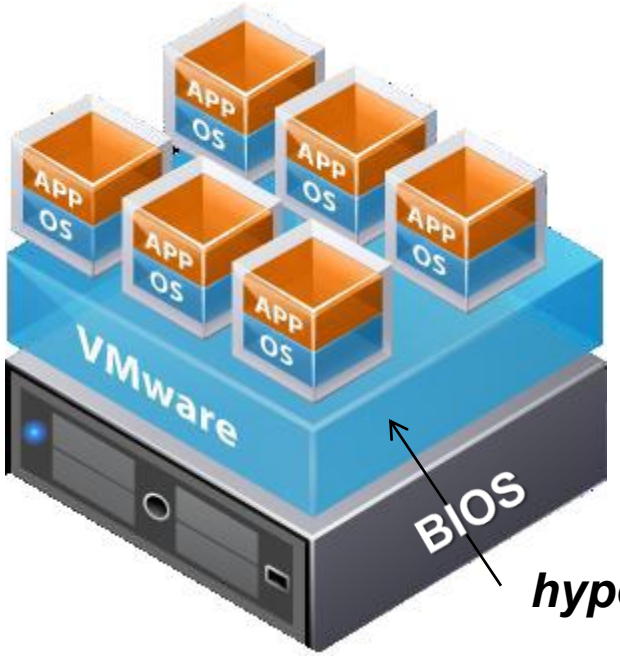


Types of Virtualization

1. HARDWARE VIRTUALIZATION
2. SOFTWARE VIRTUALIZATION
3. OS VIRTUALIZATION

How Hardware Virtualization Works

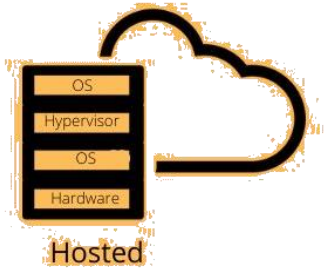




hypervisor

- In a traditional platform, the operating system divides hardware resources between applications
  - Is the go-between the BIOS and the applications
- In a virtual platform, a hypervisor divides hardware resources between operating systems

Hypervisor Types



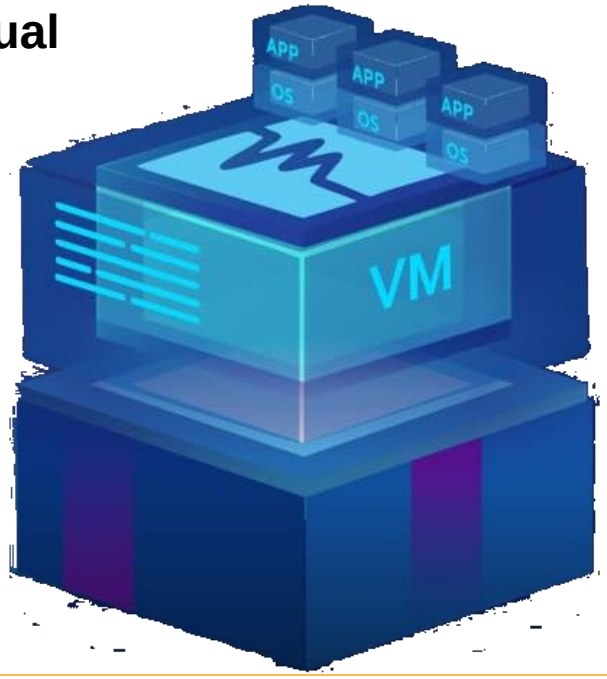
- A hypervisor, also called virtual machine monitor (VMM), allows multiple operating systems to run concurrently on a host computer— a feature called hardware virtualization
- *Type 1* (or *native*, *bare metal*) hypervisors run directly on the host's hardware to control the hardware and to monitor guest operating systems. A guest operating system thus runs on another level above the hypervisor
  - This model represents the classic implementation of virtual machine architectures; the original hypervisor was CP/CMS, developed at IBM in the 1960s, ancestor of IBM's z/VM
- *Type 2* (or *hosted*) hypervisors run within a conventional operating system environment. With the hypervisor layer as a distinct second software level, guest operating systems run at the third level above the hardware
- Microsoft Hyper-V (released June 2008) exemplifies a type 1 hypervisor (that is often mistaken for a type 2)
- Microsoft VirtualPC , Oracle Virtual Box are type 2 hypervisors

Definition

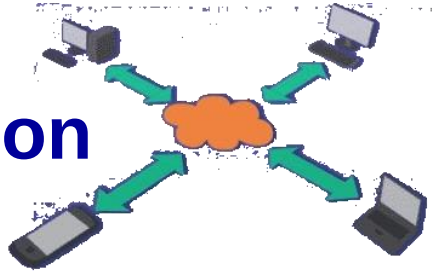
- ***Virtualization is a framework or methodology of dividing the resources of a computer into <u>multiple</u> <u>execution environments</u>..***
- ***..by applying one or more concepts or technologies such as***
  - ***Hardware and software partitioning,***
  - ***Time-sharing,***
  - ***Partial or complete machine simulation,***
  - ***Emulation***

1. Hardware Virtualization

- A computer hard disk copied into a file on another computer
  - Files can be downloaded or emailed
  - Great way to try out new operating systems
  - Great application and training test-bed
  - VMs can be distributed for real use as Virtual Appliances



2. Software Virtualization



- Encapsulation and delivery of an application and its dependencies
  - Examples include *XenApp*, *VMware ThinApp*, and *Microsoft App-V*
  - Multiple packages with conflicting dependencies can coexist
    - Dependencies mostly consist of software libraries
    - Word 97 and Word 08
- Three primary forms of delivery:
  - Packaging and installation
  - Streaming application (units of functionality)
  - Streaming presentation

3. OS Virtualization

- Where the kernel of an operating system allows for multiple isolated user-space instances, instead of just one
  - Virtualization capability part of the host OS
  - Such instances (often called *Containers* or *Jails*) may look and feel like a real server, from the point of view of its owner
  - Commonly used in virtual hosting environments, where it is useful for securely allocating finite hardware resources amongst a large number of mutually-distrusting users
  - All Guest servers *must* run *the* same OS; you can't mix and match OSs: A homogeneous environment
- Examples include Windows *Workstations*, Solaris *containers*, Parallels *Virtuozzo, Docker Containers*
- Allows maintenance of a ***single*** OS image
  - Less repeated work
  - Creates a single point-of-failure

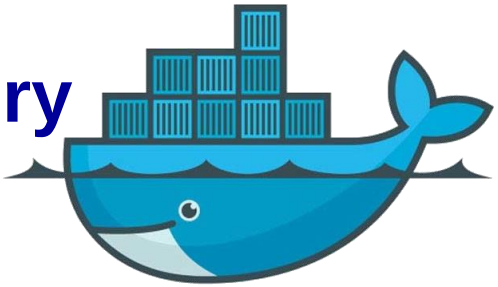
OS Virtualization Detail

- Windows (before 10) does some isolation work, but for users, not apps
- In Unix/Linux, *Containers* used to require *custom* kernels
  - .. until Docker came along (LXC project)!
  - There are six types of namespaces, which provide per-process isolation of the following operating system resources: filesystems (MNT), UTS, IPC, PID, network and user namespaces
    - By using network namespaces, each process can have its own instance of the network stack (network interfaces, sockets, routing tables and routing rules, etc.)
    - Similarly, when using the MNT namespace, when mounting a filesystem, other processes will not see this mount, and when working with PID namespaces, you will see by running the ps command from that PID namespace only processes that were created from that PID namespace
    - The CGROUPS subsystem provides resource management and accounting. It lets you define easily, for example, the maximum memory that a process may use. This is done by using CGROUPS VFS operations
      - Project was started by two Google developers, Paul Menage and Rohit Seth, back in 2006, and it initially was called "process containers"

Linux-based Containers

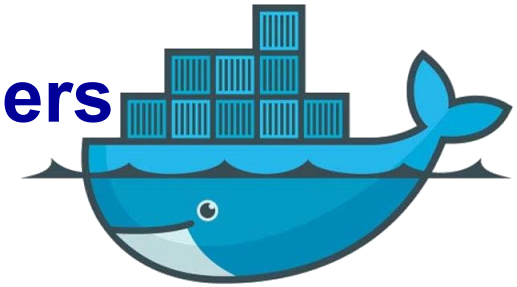
- A *Container* is an isolated user *and* application space instance on a single operating system
  - The overhead of OS virtualization is *smaller* than with server virtualization: Containers do not create another instance of the OS kernel!
  - Also, starting and stopping a Container is faster than starting or stopping a VM!
- A Container is a Linux process that has *special features* and that runs in an *isolated environment*, configured on the host (Virtual Environment (VE),Virtual Private Server (VPS))
  - *OpenVZ*: the origins of the OpenVZ project are in a proprietary server virtualization solution called Virtuozzo, which originally was started by a company called SWsoft, founded in 1997
  - In 2013, Google released the open-source version of its container stack, *lmctfy*
    - Maintainers in May 2015 stated their effort to merge their concepts and abstractions into Docker's underlying library libcontainer and thus stopped active development of lmctfy
  - *LXC Project* provides a set of userspace tools and utilities to manage Linux containers
    - Many LXC contributors are from the OpenVZ team
    - As opposed to OpenVZ, it runs on an unmodified kernel

Docker History



- **Docker started as an Open source project with the aim to automate deployment of applications inside software Containers**
  - **Docker is a tool that can *package any application and its dependencies in a virtual container than can run on any Linux Server (and now with Windows 10 on any Microsoft Server), on premise, on the private or public cloud, bare metal or not, with little to no overhead***
  - **Started as an internal project by a Platform-as-a-Service (PaaS) company called *dotCloud*, and now called *Docker Inc*. The initial prototype was written in *Python*; later the whole project was rewritten in *Go***
  - **VMWare's VMs are famous for their very low overhead, ~5%, but that is still overhead!**
  - **Using Docker to create and manage containers makes it easier to create distributed systems (and microservices) by making it super simple for multiple applications to run autonomously on a single physical machine of across a number of machines**
  - **Docker utilizes the LXC toolkit and currently available only for Linux and Windows 10**
    - **Runs on distributions like Ubuntu 12.04, 13.04; Fedora 19 and 20; RHEL 6.5 and above; Win10, and cloud platforms like Amazon EC2, Google Compute Engine and Rackspace**
  - ***Vagrant* (configures virtual dev environments), supports Docker containers natively**

Docker Containers



- Docker images can be stored on a public repository and can be downloaded with the docker `pull` command:
  - `docker pull ubuntu` or `docker pull busybox`
- Running a Fedora docker container is simple; after installing the docker-io package, you simply start the docker dæmon with
  - `systemctl start docker`
- Then you can start a Fedora docker container with
  - `docker run -i -t fedora /bin/bash`
- Changes you make in a container are lost if you destroy the container, unless you commit your changes (much like you do in git)
- You can create images by running commands manually and committing the resulting container, but you also can describe them with a Dockerfile
  - Just like a Makefile will compile code into a binary executable, a Dockerfile will build a ready-to-run container image from simple instructions
  - The command to build an image from a Dockerfile is `docker build`

Mobile Code

- **The dream of Mobile Code (runs on any platform), possible with Containers**

Docker on Windows

- ***"Quite frankly, we thought we'd leave Redmond with an agreement to make Docker for Linux run well on Hyper-V, or run well in Azure," he continued. "The first big surprise was that we were not only going to work on Docker for Linux; we were going to work on Docker for Windows. We were going to enable the four million Docker Linux developers to join all the millions of Windows developers, and make it possible using standard Docker, using open source, to take any Windows application, Docker-ize it, and run it on <u>any</u> server"***
  - **Ben Golub, Docker CEO**

URLs

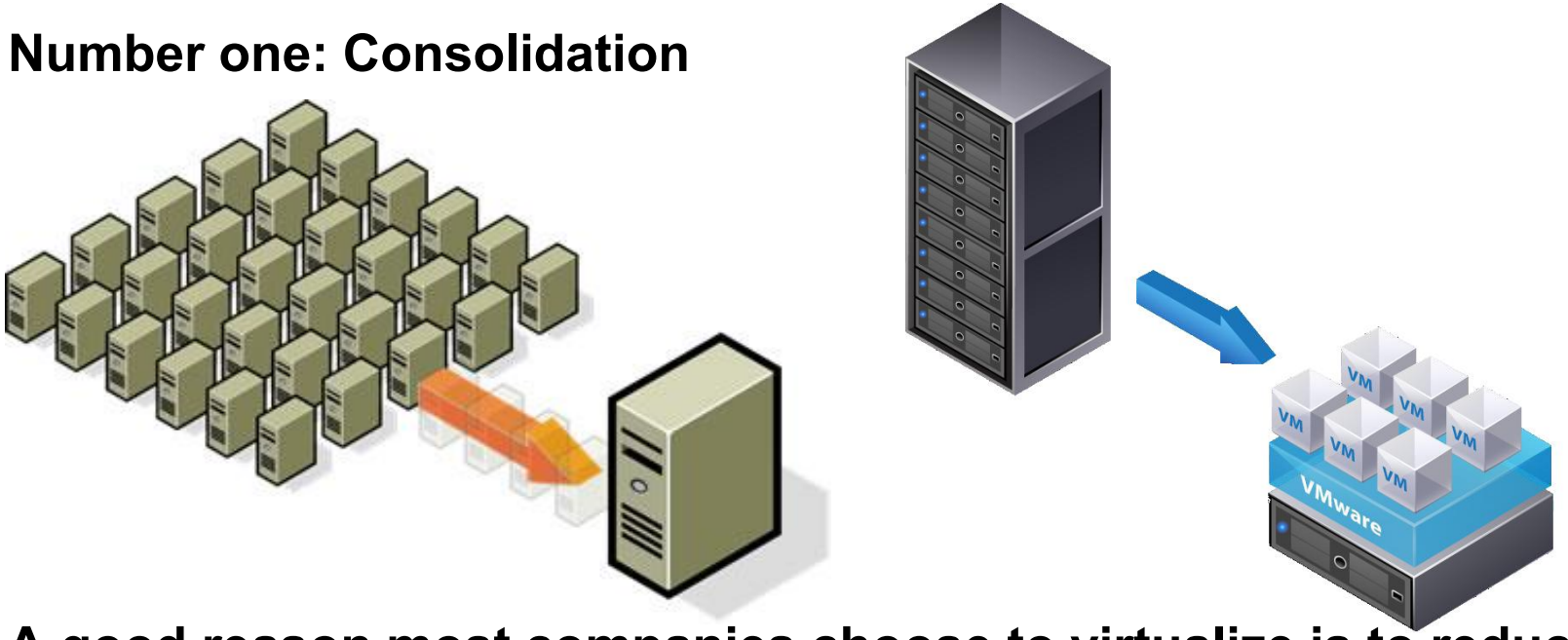
- Google Containers: https://github.com/google/lmctfy
- OpenVZ: http://openvz.org/Main_Page
- Linux-VServer: http://linux-vserver.org
- LXC: http://linuxcontainers.org
- libvirt-lxc: http://libvirt.org/drvlxc.html
- Docker: https://www.docker.io
- Docker Public Registry: https://index.docker.io
- Docker about: http://www.linuxjournal.com/content/linux-containers-and-future-cloud

Virtualization Public Cloud Benefit: Instant Scalability

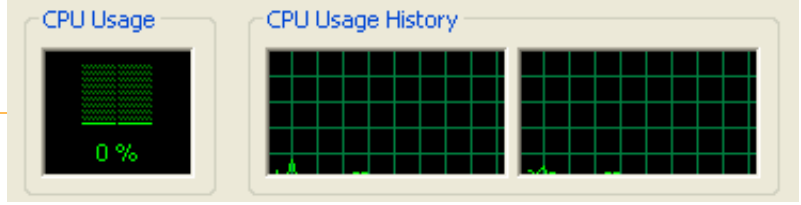
- ***Winuxdroid*** **does not** ***scale*****! So instead, we build** ***a farm of application clones*****, with each app running on a** ***separate*** **OS, all connected to a front-facing load balancer**

Virtualization Back Office Benefit: Cost reduction

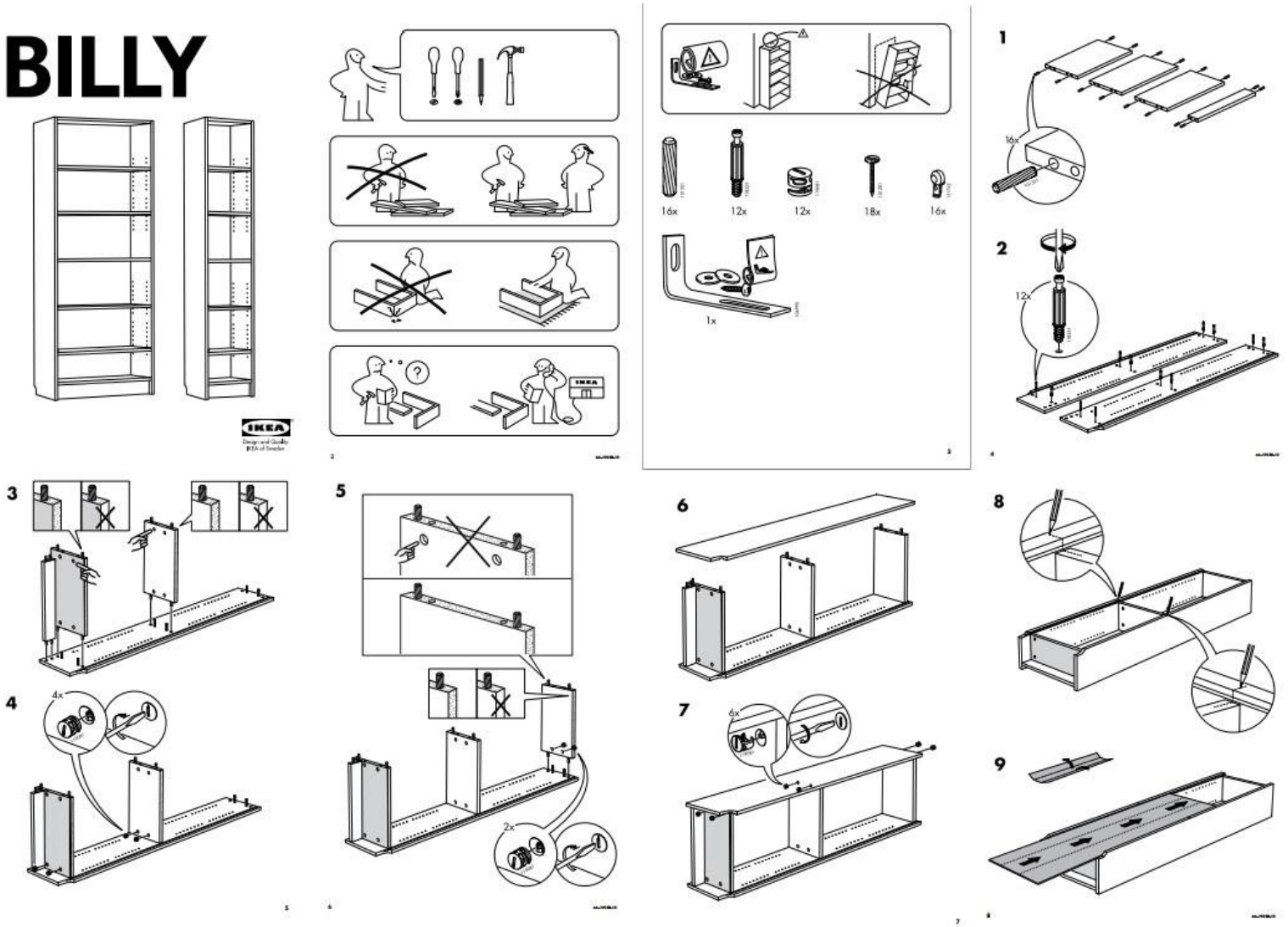
- Number one: Consolidation



- A good reason most companies choose to virtualize is to reduce cost through *consolidation*
- Virtual machines can be used to consolidate the workloads of several under-utilized servers to fewer machines, perhaps a single machine



Virtualization App Benefit: No installation required!

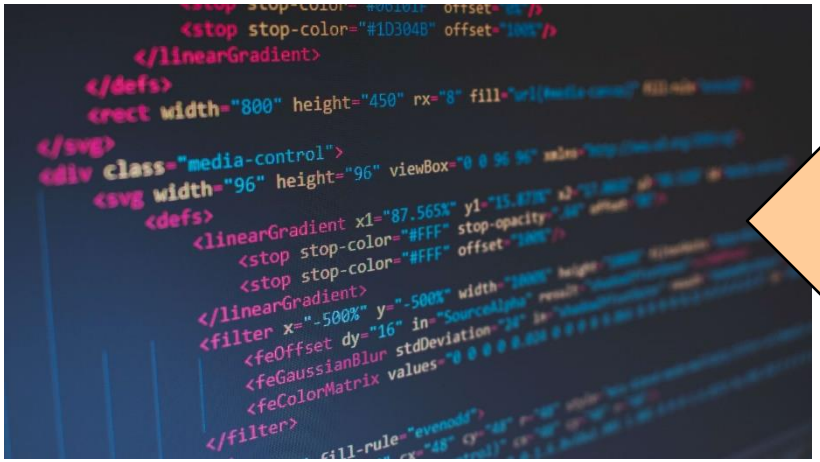


Software engineering productivity benefit

- Once I create an entire stack of applications and servers, I can actually save that entire stack as a container image
- Then I can restart that same container image on any computer
- That means, I don't have to do any installations, or testing, or configuration modification
  - Because the binary code is the same, running on the same kind of (either Windows or) Linux VM

Dockerhub..

- .. Is like Github for *apps*!



GitHub

docker hub

Thread, process, container

- Threads are lighter-weight than processes which are lighter-weight than containers (by a small margin)

Namespaces and control groups

- **Namespaces** were originally a unique feature for Linux providing a way to control what resources with which a process can interact
- They are quite different from access controls because the process doesn't know the resources exist
  - A simple example of this is the process list: there could be 100 processes running on a server, yet a process running within a namespace might see only 10 of those. Another example might be for a process to think it's reading from the root directory when in fact it has been virtualized
- In 2006, the Linux kernel was added the support for grouping processes together under a common set of resource controls in a feature called **cgroups**
- It's the combination of **cgroups** and **namespaces** that became the foundation of modern-day containers

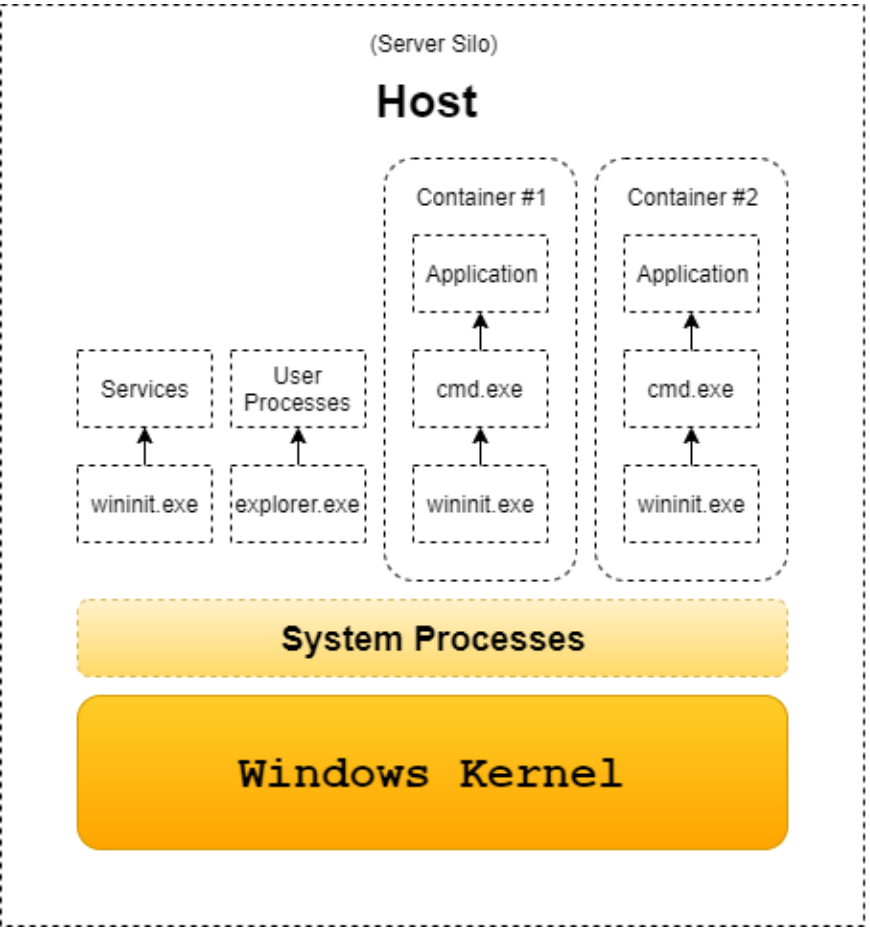
On Windows

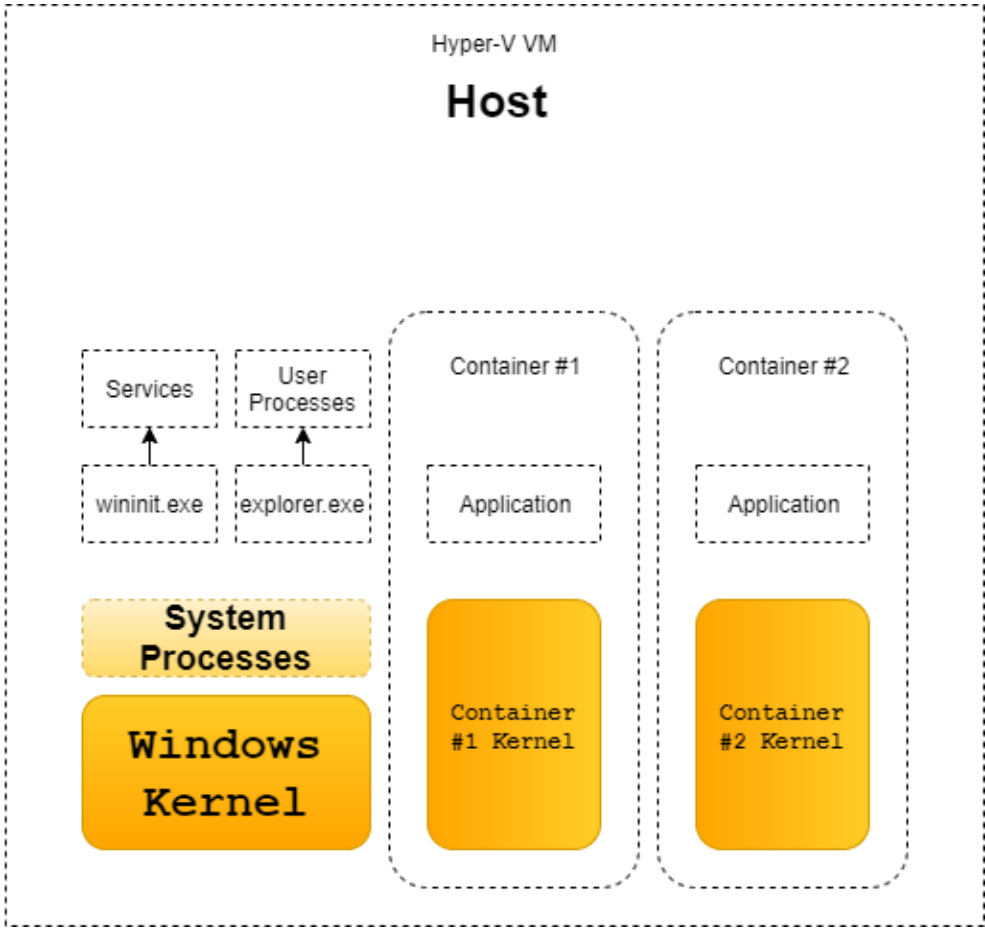
- Windows already had a control groups-like feature called **job object**. A job object allows groups of processes to be managed as a single unit. Examples include enforcing limits such as working set size and process priority or terminating all processes associated with a job. Microsoft was still missing a namespace-like feature, and so a kernel object called a **silo** was born
- Microsoft introduced two types of container architectures:
  - Deploying an application in a fully isolated, Hyper-V virtual machine, which is supported on both Windows 10 and Windows Server.
  - Deploying an application in a lightweight silo container, which is currently supported only in Windows Server. Server silos allows Windows to have isolation good enough for a full container solution
- Most of the named objects in Windows (files, registry, symlinks, pipes) are under a root namespace called the root directory object
- When a silo is created, another root directory object is created for that silo. From that point on, there is a different value for any named object (for example, the symbolic link "`C:`")

On Windows

- There are plenty of functions in the Windows kernel used to distinguish between processes and threads inside a container from those which are outside
- None of the checks related to identifying a containerized process happen in user mode. The kernel is responsible for distinguishing between operations of regular processes and container processes, including access to files
- Unlike Linux, applications don't use system calls directly but rather call API functions that are delivered through internal DLLs. Some of those DLLs are not documented. Those functions call other API functions and which may themselves call others until eventually, the system call occurs
- Once we arrive to kernel land, our first stop is **`NtCreateFile`** . This function doesn't do much by itself and simply calls **`IopCreateFile`**. In **`IopCreateFile`** the kernel starts handling silos by first checking if the current thread is attached to a silo

On Windows





Question

- Let's take a μservice app made up of the following two components:

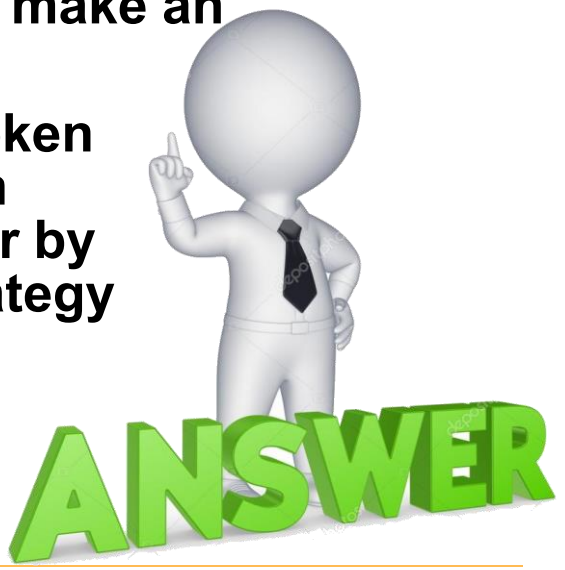
- A front-end application built on a UI Web Framework
- A back-end REST API server built on Java EE
- This application historically ran in a single Tomcat server and the two components were communicating over a REST-based API

- Should I break this application into multiple containers?

Answer

- Yes, this application should be decomposed into two different Docker containers... but only after careful consideration

- Instead of breaking applications up into multiple containers *just because* or in an attempt to adhere to some newfangled principle (e.g. "run only one process per container"), think this through the engineering requirements to make an informed and intelligent decision
- Whether or not all applications should be broken into multiple containers - containerizing them should, at the very least, make your life easier by providing you with a simpler deployment strategy



Historically

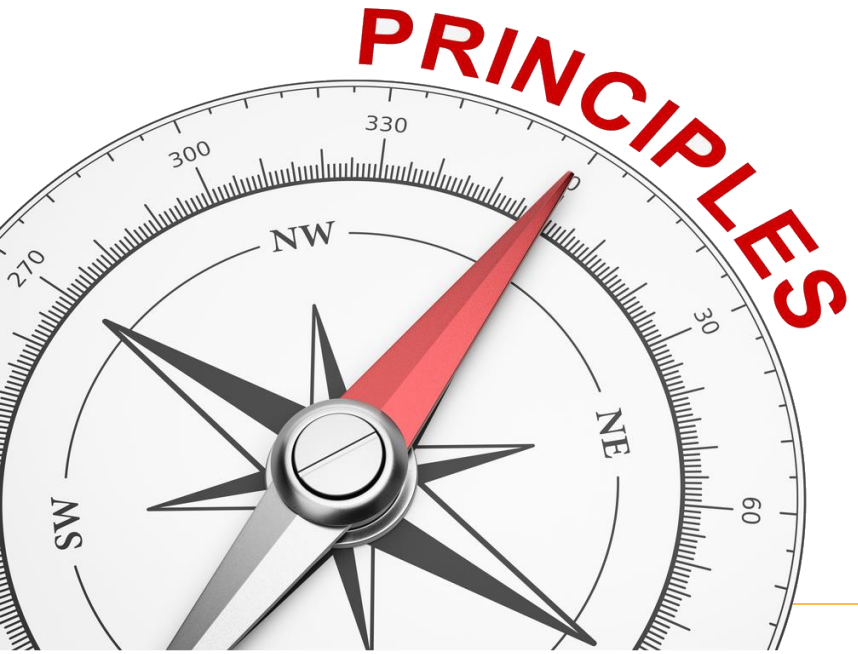
- The JVM is multi-threaded, so you are not necessarily running multiple Unix/Linux processes
  - Historically, Java liked multiple applications running in the same JVM; doing so can in practice, save memory at scale
    - Web application servers like Tomcat were built from the ground up to support running multiple applications in a single JVM. That's actually one of the main differences between running a simple Java program versus a Java EE application
- In reality many apps use multiple processes per container
- Modern web applications featuring event driven programming and fire off many sub processes
- Basic idea is to offload work to other processes (or threads) and let the kernel handle the I/O
  - Linux kernel is quite good at scheduling sub processes
  - Kubernetes/Swarm and individual Docker containers are not as good at this
  - Processes (and threads) are about kernel resource allocation; containers are about cluster resource allocation.

Observations

- The two components seem to be doing different things
  - One component is a web front-end, the other is an API server. Since these components are doing different things (i.e they are indeed different services), there is little chance that there would be a performance benefit from being in the same JVM
- These two applications communicate using a REST API (...instead of using sockets, shared memory or files, etc.)
- Generally, if an application contains an API layer and a front-end layer, it's useful to scale these independently
  - For example, if the API is also consumed by a mobile application it might be useful to scale it up and down with user load, where the web front-end may not need to be scaled.
  - Conversely, if I scale the web front-end, I may also need to scale the API server portion, but I may only need one more API server for every five web front-ends
- Scaling logic can be complicated and being able to scale these independently with Kubernetes could be very useful

Containerization principle

- If your application / service has *good separation of code, configuration, and data*, installs cleanly (as installer scripts can make this whole process difficult), and features a clean communication paradigm - it does make sense to break the application up / allocate one service per container
- Containerization is more than just philosophy, it's about solving technical pain(s)



References

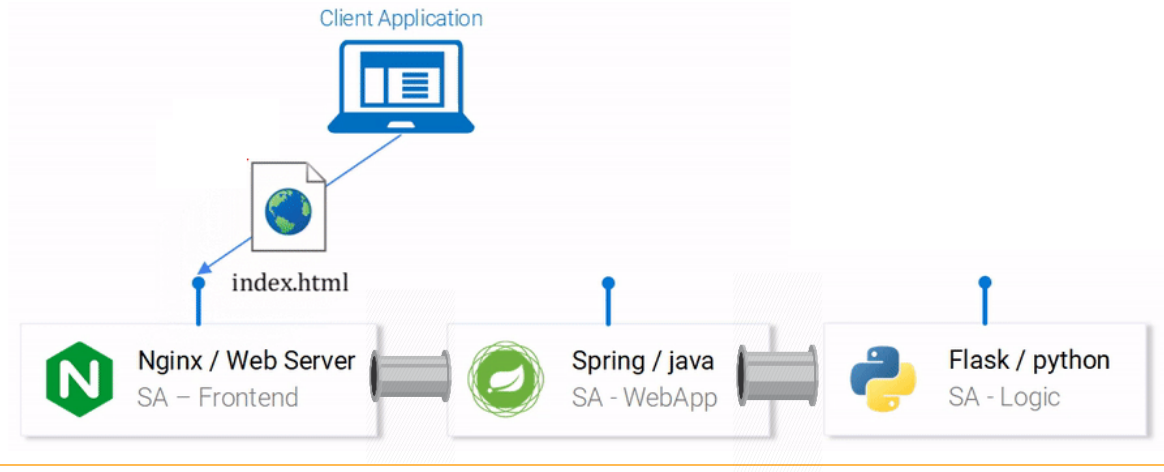
- https://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers/

Goal for today

- Build a container image for each service of our μservice-based apps
  - You debugged yours and it works, right? Otherwise, please work with TA
- *Next week get ready to i*ntroduce Kubernetes and deploy the μservice-based app into a Kubernetes Managed Cluster

The μservice-based app

- **`SA-Frontend`:** Nginx web server that **serves out ReactJS** static files
- **`SA-WebApp`:** Java Web App that **handles requests** from the frontend
- **`SA-Logic`:** a python application that **performs Sentiment Analysis**
- Very important part of bot-based enterprise apps, where capturing sentiment is critical to a smooth-running enterprise



Web-based Flow

- Browser requests index.html (bundled ReactJS app)
- User triggers requests to a Spring-based WebAPI app
- Spring WebAPI forwards requests for sentiment analysis to Python app
- Python app calculates sentiment and returns the result as a response
- Spring WebAPI returns response to the React app, which then presents the information to the user
- *All services will communicate with Web-based (HTTP/HTTPS) channels*
  - *Not always true: Often μservice apps communicate by pushing messages instead of pulling HTML/XML/JSON*
  - *A very popular way to communicate todays is push-based (rather than pull-based) using message broker framework Apache Kafka*
  - *https://kafka.apache.org/*

Note

- I use *docker toolbox*, an older version of docker that harkens back to the days we had to run a hypervisor and create a VM in order to run containers in this VM
- When this because possible, Cupertino and Redmond became very very scared that everyone was going to start using Linux *more*, Macs and Windows *less*
- So they paid Docker to develop a docker edition that runs with the Mac OS and the Window OS instead of a separate Linux VM. That version is called *Docker Desktop*
  - But of course, a Mac OS is POSIX compliant
  - And in Windows you can run the Windows Subsystem for Linux (WSL)

192.168.99.100

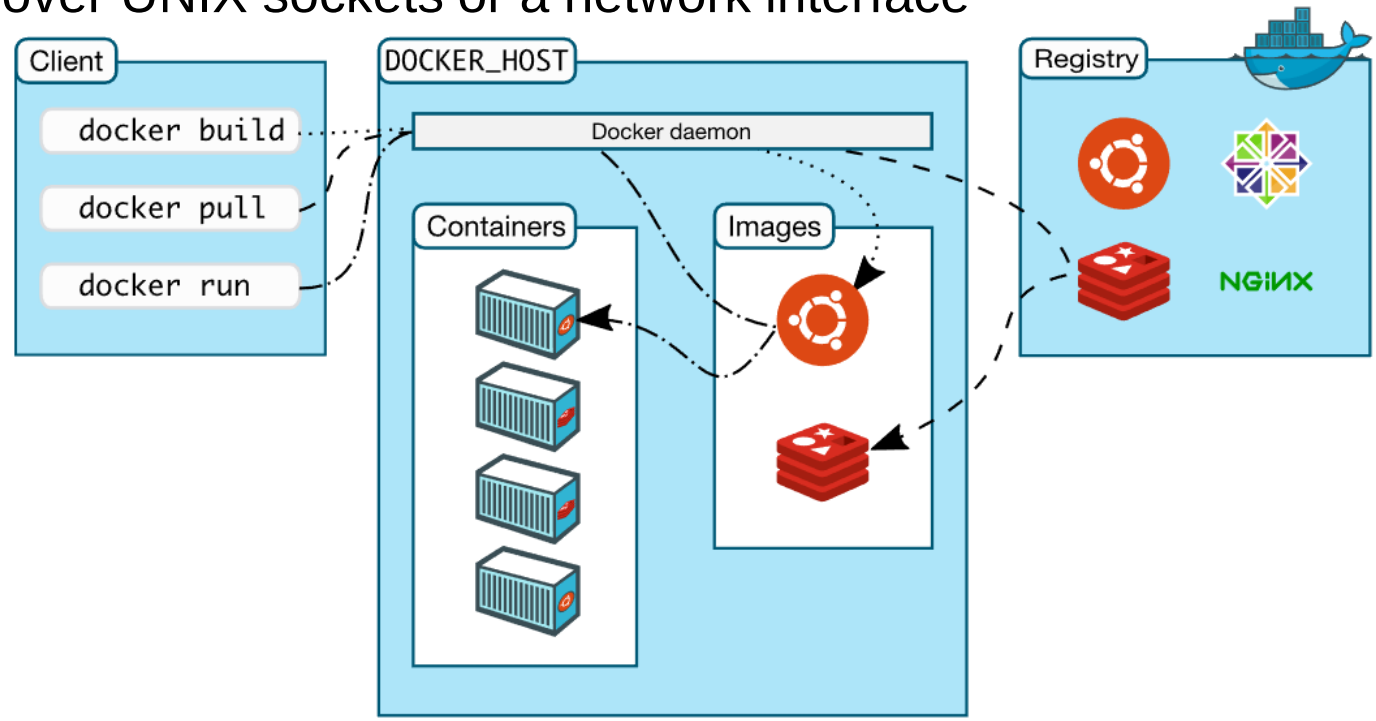
- When you see 192.168.99.100 in these slides, that is the IP of my Virtualbox-powered Linux VM
- If you're going to be using *Docker Desktop*, replace that URL with localhost

Part 3

BUILDING CONTAINER IMAGES WITH DOCKER

Docker architecture

- The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers
- The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon
- The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface

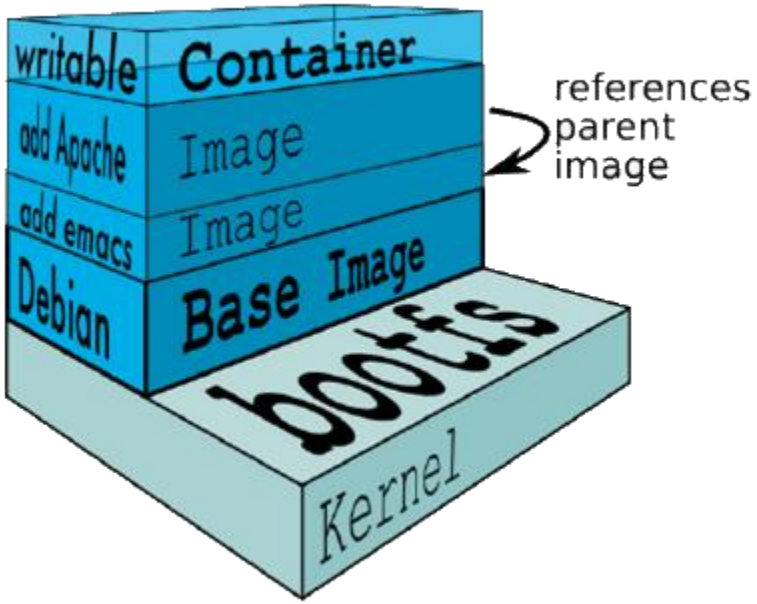


Container *image* (blueprint)

- **A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings**
- **Available for both Linux and Windows based apps, containerized software *will always run the same, regardless of the environment***
- **It means that containers can run on *any* computer — even in the production server — with no differences**
- The basic building block for a Docker container is the **`Dockerfile`**
  - The **Dockerfile** starts with a base container image and follows a sequence of instructions on how to build a new container image that meets the needs of your app
  - So a **DockerFile** is a sequence of installation instructions, like the ones we programmed for **wordpress**

Container image

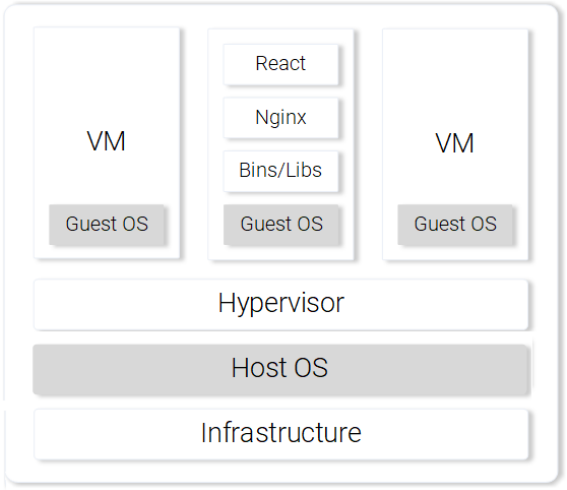
- An image is a collection of files and some metadata
- Images are comprised of multiple layers, multiple layers referencing/based on another image (Union File System)
- Each image contains software you want to run
- Every image contains a base layer
- Layers are read-only



Containers vs. VMs

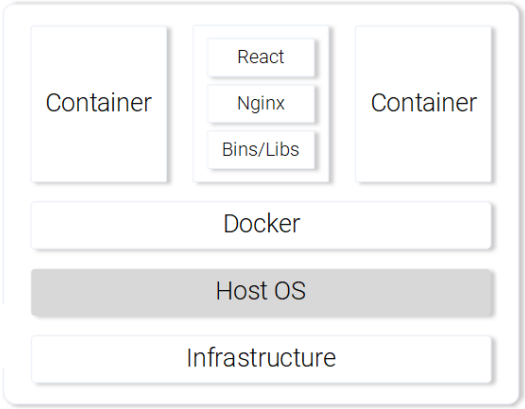
Cons of using a VM

- Resource inefficient, each VM has the overhead of a fully-fledged OS
- Platform dependent. What worked on your computer might not work on the production server
- Heavyweight and slow scaling when compared to Containers



Pros of using a Container

- Resource efficient, use the Host OS with the help of Docker
- Platform *independent*. The container that you run on your computer will work *anywhere*
- Lightweight using image layers



Docker *stuff*

- ***Docker Client*** **is the user interface that allows communication between the user and the Docker daemon**
- ***Docker Daemon*** **sits on the host machine answering requests for services**
- ***Docker Hub*** **is a centralized registry allowing backup of Docker container images with public and private access permissions**
- ***Docker Containers*** **are responsible for the actual running of applications and includes the operating system, user added files, and meta-data**
- ***Docker Images*** **are ready-only templates that help launch Docker containers**
- ***DockerFile*** **is a file housing instructions that help automate image creation**

Install Docker Desktop (or Toolbox)

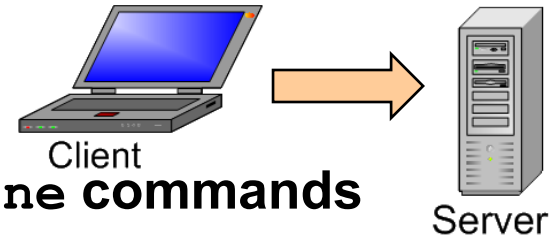
- If Mac hardware new, Mac OS new, >=4G RAM, then install **`Docker for Mac`**:
  - https://docs.docker.com/docker-for-mac/install/
- Windows 10 (Professional and above) and Windows Server 2016, >=4G RAM, BIOS-enabled virtualization, CPU-SLAT capable, have native support for Docker with Hyper-V containers
  - https://docs.docker.com/docker-for-windows/install/
- If you *don't* have Windows 10 Professional or don't want to play around with BIOS settings, or you're an older MAC, then install **`Docker Toolbox`**, which uses Oracle Virtual Box virtualization:
  - https://docs.docker.com/toolbox/overview/
  - My Oracle VirtualBox install is at **`C:\Program Files\Oracle\VirtualBox\VirtualBox.exe`**
  - My Docker toolbox install is at **`D:\user\DockerToolbox`**

docker is configured to use the default machine with IP 192.168.99.100

Running docker toolbox

- **Start Docker Toolbox by running `start.sh`**
  - **For me: `D:\user\DockerToolbox\start.sh`**
  - **Will start git shell**
  - **Which will start your Oracle Virtualbox, *make sure your external USB drive is connected if that is where yout Oracle Virtualbox creates VMs, because dockertoolbox will create a linux VM to host all its containers***
- docker is configured to use VM called ***default*** with IP **192.168.99.106**
- **Docker desktop for Windows/Mac runs on *localhost* machine**

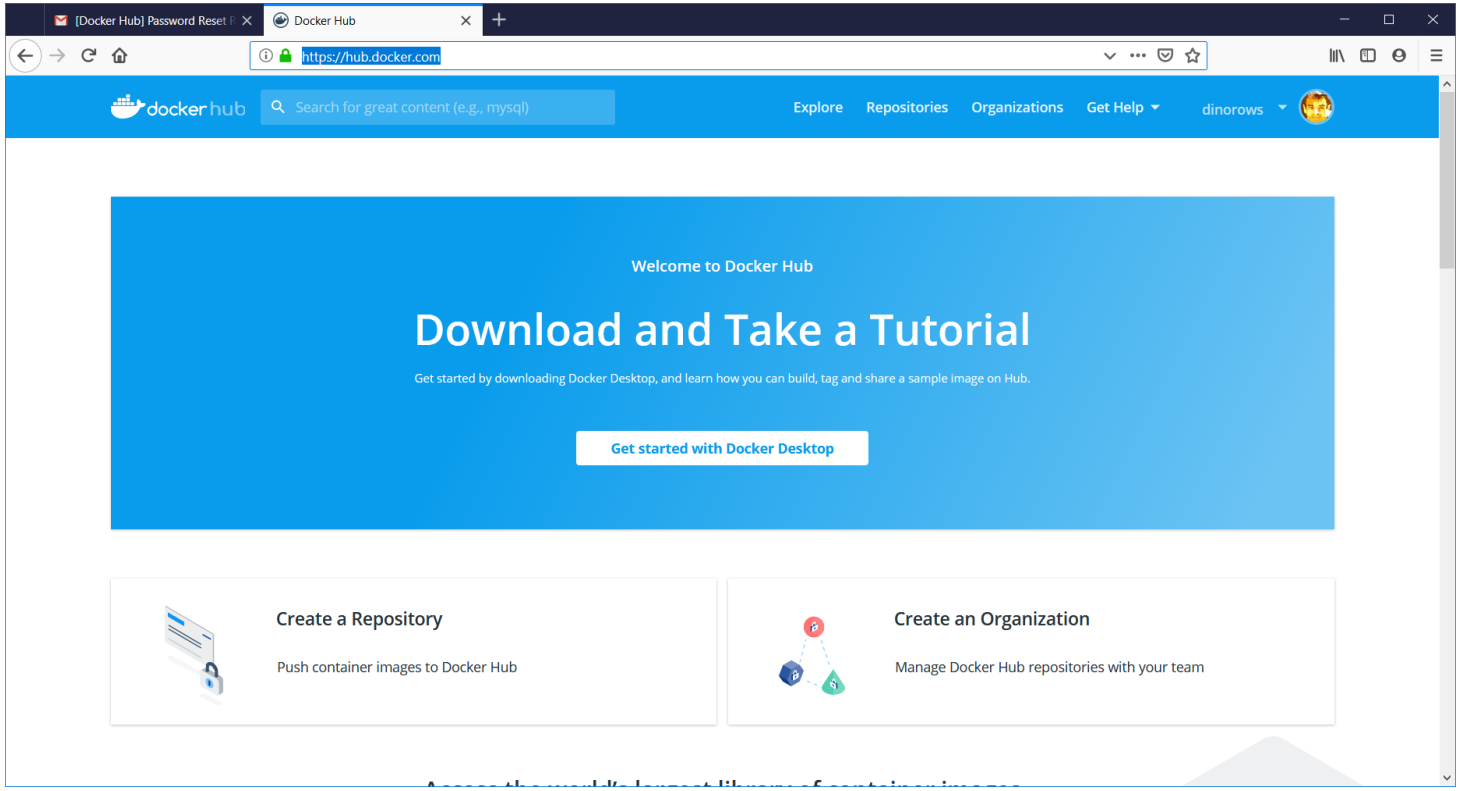
Docker Toolbox



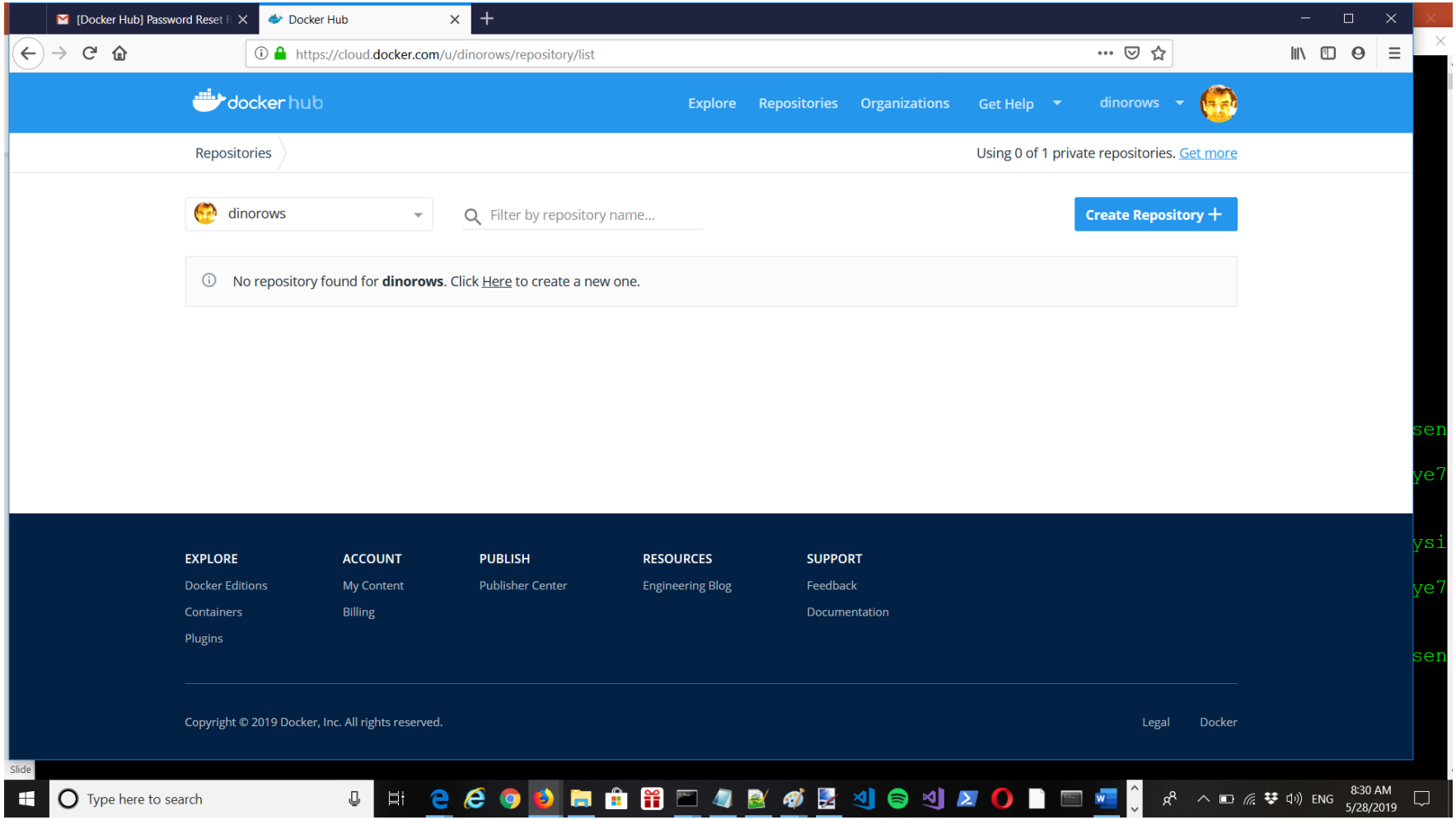
- **Toolbox includes these Docker tools:**
- ***Docker Machine* for running `docker-machine` commands**
  - Docker Machine is a **tool that lets you install Docker Engine on virtual hosts**, and manage the hosts with docker-machine comman
  - You can use Machine to create Docker hosts on your local Mac or Windows box, on your company network, in your data center, or on cloud providers like Azure, AWS
- ***Docker Engine* for running `docker` commands**
- ***Docker Compose* for running `docker-compose` commands**
  - **Docker Compose allows you to compose Containers, like we did with `wordpress` and `mysql` containers when we deployed a blog engine on Azure using Azure templates, remember?**
- ***Kitematic*, the Docker GUI**
- **A shell preconfigured for a Docker command-line environment**
- ***Oracle VirtualBox***

Get an account with DockerHub

- Register here: https://hub.docker.com/
- You will need to create a docker ID, docker password, and an email

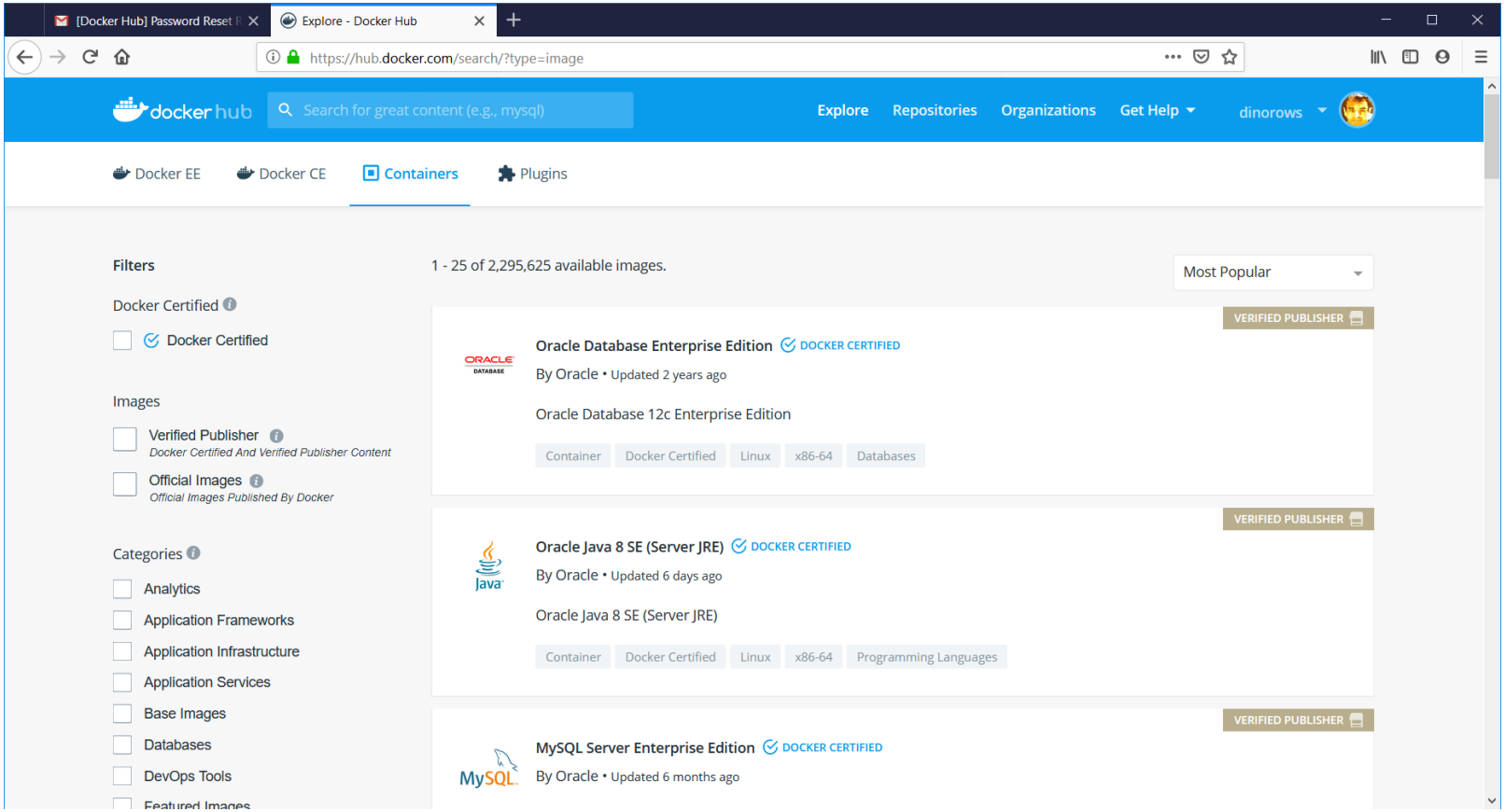


No private repositories, *yet*



Public repositories

- Just like AWS & Azure marketplaces!



Check versions

- `(sudo) docker --version`
  `docker version 18.09, build c97c6d6`
- `docker-compose --version`
  `docker-compose version 1.24.0, build 8dd22a9`
- `docker-machine --version`
  `docker-machine version 0.16.0, build 9ba6da9`

`hello-world` Container *image* to Container *instance*

- **`docker run hello-world`**

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

Ubuntu bash shell

- Mac: **`docker run --interactive --tty ubuntu bash`**
- Windows: **`winpty docker run --interactive --tty ubuntu bash`**
- **`ls`**

```
root@a53bd7b99908:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
```

- **Check the hostname of the container:**
  - **`Hostname`**
    hostname is assigned as the container ID
- **Exit the shell with the `exit` command (which also stops the container) :**
  - **`exit`**

Alpine linux

- Mac: `docker run -it alpine env`
- Windows: `winpty docker run -it alpine env`
  - `winpty docker run -it alpine sh`

https://alpinelinux.org/

Docker Container image and instance mgt.

- **Stop and remove containers and images with the following commands. Use the "all" flag (--all or -a) to view stopped containers.**
- `docker container ls`
- `docker container stop webserver`
- `docker container ls -a`
- `docker container rm webserver`
- `docker image ls`
- `docker image rm nginx`

Docker networking

- `docker network ls`

Part 4

BUILDING OUR OWN CONTAINER IMAGES

Task 1: Building Container image for React app: Build the `Dockerfile`

- Steps we took to serve the react static files using nginx:
  - Build the static files (**`npm run build`**)
  - Copy the contents of the **`build`** folder from your **`sa-frontend`** project over to **`nginx/html`**
  - Startup the **nginx** server
- The instructions in the Dockerfile for the **`sa-frontend`** project is only a two-step task because the Nginx Team provided us with a base image for Nginx, which we will use to build on top off
  - Start from the base **`nginx`** **Image**
  - Copy **`sa-frontend/build`** folder to the container's **`nginx/html`** folder
- So, create a file called **`Dockerfile`** in **`sa-frontend`**:

```
FROM nginx
COPY build /usr/share/nginx/html
```

  - As described in nginx's dockerhub documentation!
    - https://hub.docker.com/_/nginx/

Detail

- In the latest nginx docker image, the config file is in folder `/etc/nginx`, and the `index.html` file is in folder `/usr/share/nginx/html`
- Status of nginx server in docker container may be `not started`
  - Make sure to run `docker run` cmd to start nginx server

Update React source to access java webapp container ip (`localhost` on Docker Desktop)

- If you change the port for the **`sa-webapp`** container, or if you are using docker-machine ip (as we are since we are using docker toolbox), we need to update **`App.js`** in **`sa-frontend`** in the method **`analyzeSentence`** to fetch from the *new* IP or Port
- **`Docker-machine ip`**
  `192.168.99.100`

```
analyzeSentence() {
    fetch('http://192.168.99.100:8080/sentiment', {
        method: POST ,
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({sentence: this.textField.getValue()})
    })
        .then(response => response.json())
        .then(data => this.setState(data));
}
```

What did I do?

- **Added a *new* button, keeping the old one for `localhost` testing, and *two* handlers (`app.js`):**

```
analyzeSentenceLocal() {
   fetch('http://localhost:8080/sentiment', {
      method: 'POST',
      headers: {
         'Content-Type': 'application/json'
      },
      body: JSON.stringify({sentence: this.textField.getValue()})
   })
      .then(response => response.json())
      .then(data => this.setState(data));
}

analyzeSentence() {
   fetch('http://192.168.99.100:8080/sentiment', {
      method: 'POST',
      headers: {
         'Content-Type': 'application/json'
      },
      body: JSON.stringify({sentence: this.textField.getValue()})
   })
      .then(response => response.json())
      .then(data => this.setState(data));
}
```

Two buttons (`app.js`)!

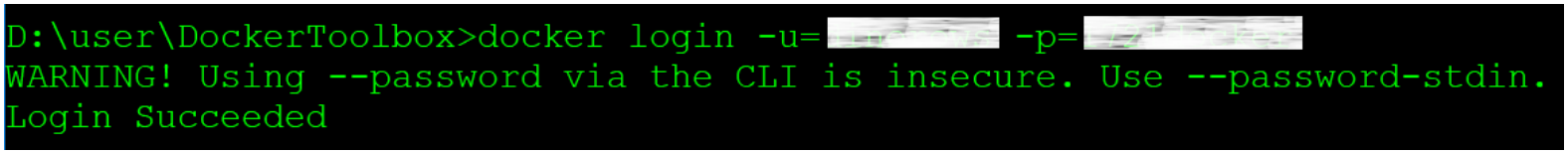
```
render() {
   const polarityComponent = this.state.polarity !== undefined ?
      <Polarity sentence={this.state.sentence} polarity={this.state.polarity}/> :
      null;

   return (
      <MuiThemeProvider>
         <div className="centerize">
            <Paper zDepth={1} className="content">
               <h2>Sentiment Analyser</h2>
               <TextField ref={ref => this.textField = ref} onKeyUp={this.onEnterPress.bind(this)}
                     hintText="Type your sentence."/>
               <RaisedButton  label="Send" style={style} onClick={this.analyzeSentence.bind(this)}/>
               <RaisedButton  label="SendLocal" style={style} onClick={this.analyzeSentenceLocal.bind(this)}/>
               {polarityComponent}
            </Paper>
         </div>
      </MuiThemeProvider>
   );
}
```

Rebuild target

- `npm run build`
  - Will create a new `build` folder

Login to Docker Hub in terminal

- **Since we're in the *Navy (CSYE 7220)*, we're instrumenting all actions through a command terminal, because machines don't care about the GUI**
  - **Notice how we ran our Springboot app using the terminal, rather than our beloved VSC?**
  - **If we were in the *Air Force (INFO 6250)*, we'd be using VSC!**
- **Login by executing the below command in your Terminal:**
  - `winpty docker login -u=<your-DOCKER-USERNAME> -p=<your-DOCKER-PASSWORD>`



docker build

- navigate to the directory `sa-frontend`. Then execute the below command (replace $DOCKER_USER_ID with your docker hub username
  - `docker build -f Dockerfile -t <your-DOCKER_USER_ID>/sentiment-analysis-frontend .`
  - Notice the "." in the above command!
  - Actually, drop "`-f Dockerfile`" from command above because you are already in the directory containing the Dockerfile!

```
D:\user\docs\NU\_Csye7220\Lecture 9 (kube)\labs\sentiment-dino\sa-frontend>docker build -f Dockerfile -t dinorows/se
timent-analysis-frontend .
Sending build context to Docker daemon  154.2MB
Step 1/2 : FROM nginx
 ---> c82521676580
Step 2/2 : COPY build /usr/share/nginx/html
 ---> 1481ed4816e2
Successfully built 1481ed4816e2
Successfully tagged dinorows/sentiment-analysis-frontend:latest
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and dire
tories added to build context will have '-rwxr-xr-x' permissions. It is recommended to double check and reset permis
ions for sensitive files and directories.
```

- *The build will take a looooong time..*

Why did the build take so long? & `.dockerignore`

- Building the image for `sa-frontend` was slow because of the build context that had to be sent to the docker daemon!
- The build context directory is defined by the last argument in the docker build command (the trailing dot), which specifies the build context. And in our case, it included:

```
D:\user\docs\NU\_Csye7220\Lecture 9 (kube)\labs\sentiment-dino\sa-frontend>ls
Dockerfile          dino-readme.txt    package-lock.json  public
build               node_modules       package.json       src
```

- But the only data we need is in the `build` folder! Uploading anything else will be a waste of time!
- Add a `.dockerignore` file:

```
node_modules
src
public
```

  The `.dockerignore` file should be in the same folder as the `Dockerfile`

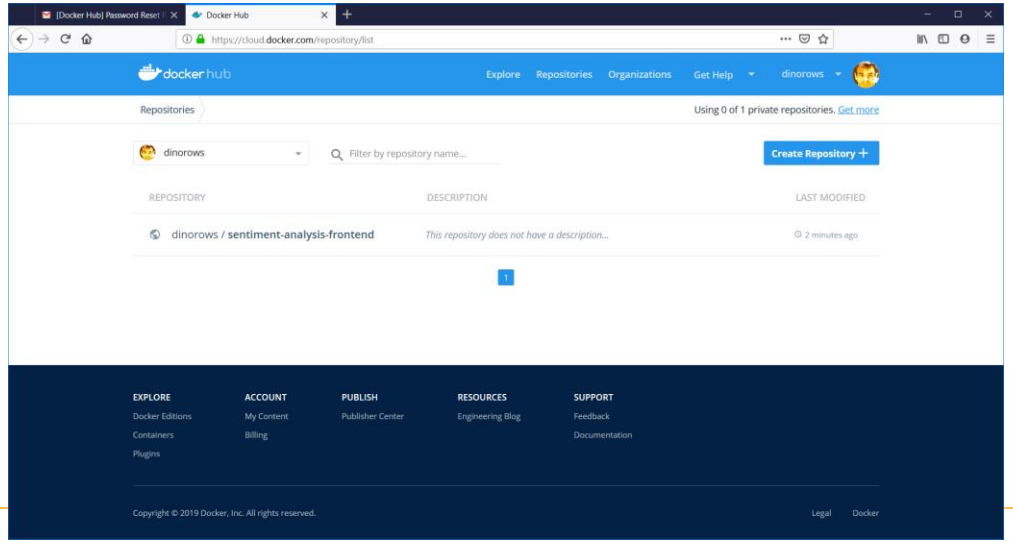
- Now building the image takes only seconds!
  - `docker build -t dinorows/sentiment-analysis-frontend .`

Push the image to *your* docker hub

- To push the image, use the `docker push` command:
  - `docker push <your-DOCKER_USER_ID>/sentiment-analysis-frontend`

```
D:\user\docs\NU\_Csye7220\Lecture 9 (kube)\labs\sentiment-dino\sa-frontend>docker push dinorows/sentiment-analysis-f
ontend
The push refers to repository [docker.io/dinorows/sentiment-analysis-frontend]
eda4989eb10f: Pushed
08d25fa0442e: Mounted from library/nginx
a8c4aeeaa045: Mounted from library/nginx
cdb3f9544e4c: Mounted from library/nginx
latest: digest: sha256:a7b480eb52d296a5984aef5d610b9e3d32f6bbf3d5d91166c56cfa7a38b9bfce size: 1158
```

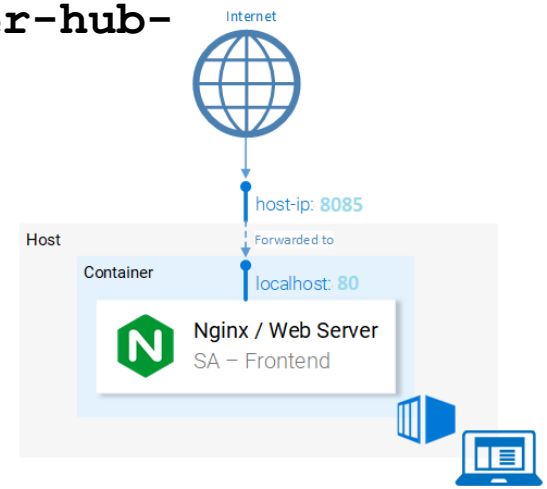
- Verify in your docker hub repository that the image was pushed successfully



Test your front-end Container image

Dino: read `readme.md` in `sentiment-dino` lab folder

- **Now the image in `<your-docker-hub-userid>/sentiment-analysis-frontend` can be pulled and run by anyone!**
- **In cmd shell:**
  - **`docker -v`**
  - **`docker image ls`**
    - **Is `<your-docker-hub-userid>/sentiment-analysis-frontend` listed?**
    - **If not, then:**
      - **`docker pull <your-docker-hub-userid>/sentiment-analysis-frontend`**
  - **`docker run -d -p 8085:80 <your-docker-hub-userid>/sentiment-analysis-frontend`**
  - The first 8085 is the port of the host (your laptop)
  - The second 80 stands for the container port to which the calls should be forwarded
    - Note that this port is different than the port you may have used when running nginx in native mode (I had modified conf/nginx.conf to listen on port 8081)



Note

- **If you test the app by writing a sentence and clicking on the button, nothing will happen because the containers for the backend sa app and web app *have not been built and are not running yet*!**

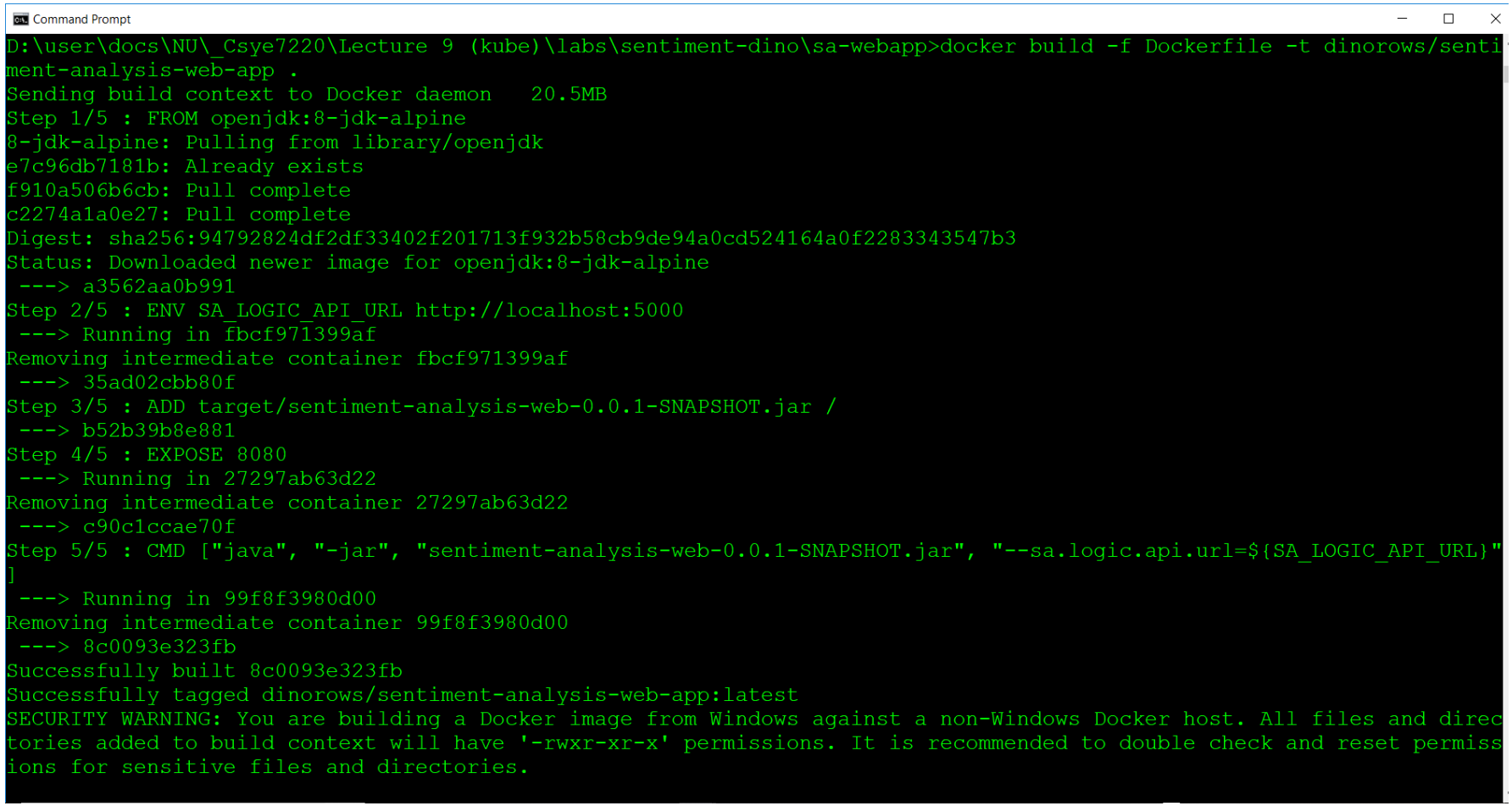
Task 2: Build container image for the Springboot app

- **Write the following `Dockerfile` in `sa-webapp` folder:**

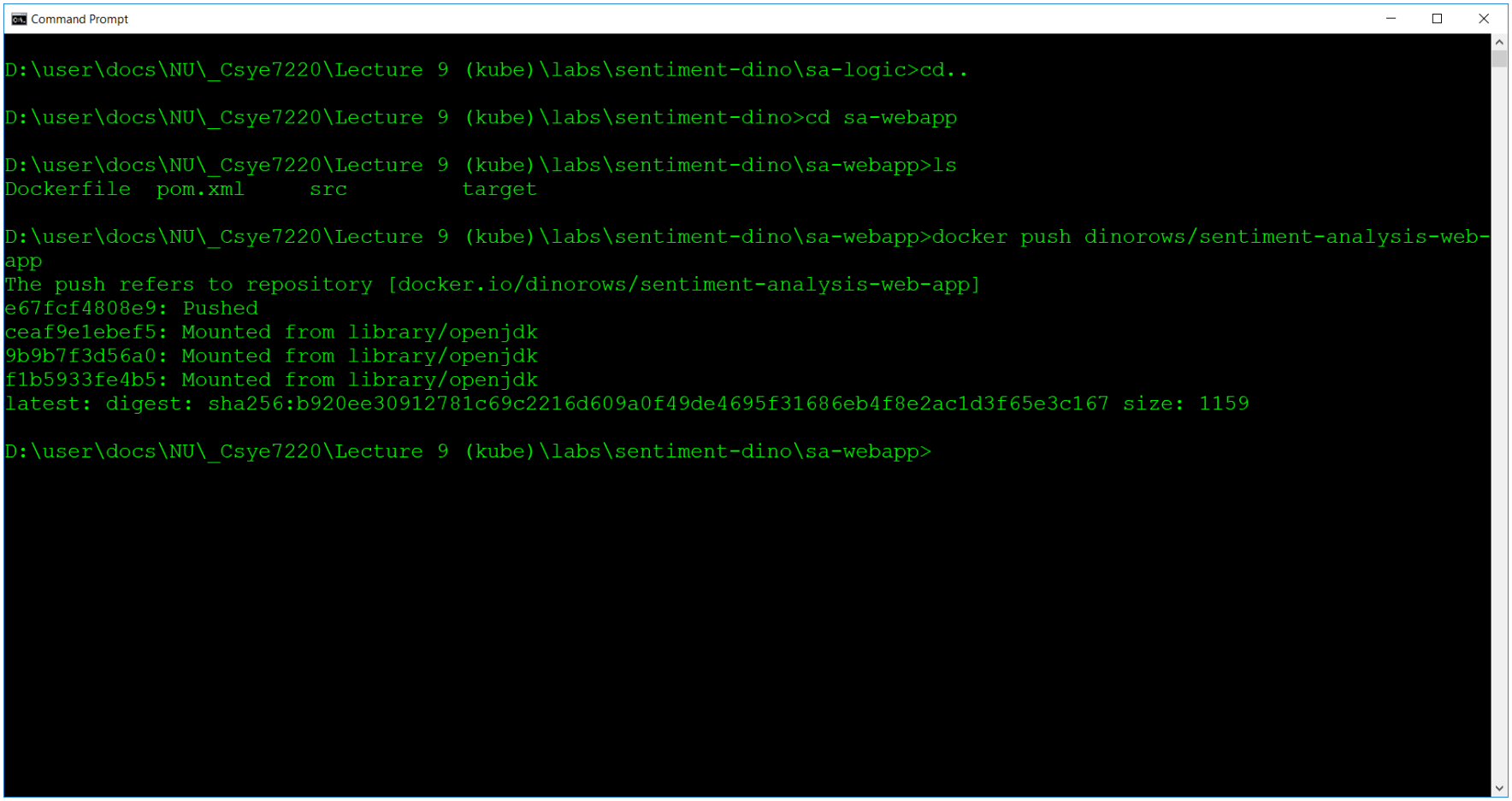
```
FROM openjdk:8-jdk-alpine
# Environment Variable that defines the endpoint of sentiment-analysis python api:
ENV SA_LOGIC_API_URL http://localhost:5000
ADD target/sentiment-analysis-web-0.0.1-SNAPSHOT.jar /
EXPOSE 8080
CMD ["java", "-jar", "sentiment-analysis-web-0.0.1-SNAPSHOT.jar", "--sa.logic.api.url=${SA_LOGIC_API_URL}"]
```

- **We will base our container on the alpine linux container with a JDK8 install**
- **We define the `SA_LOGIC_API_URL` environment variable**
- **We add the jar file to our container image**
- **We start the jar file using the `CMD` command, passing in all required arguments**
- **Build: `docker build -f Dockerfile -t <your-_DOCKER_USER_ID>/sentiment-analysis-web-app .`**
- **Push: `docker push <your-DOCKER_USER_ID>/sentiment-analysis-web-app`**

docker build



docker push



Task 3: Build container image for the python app

- **Write the following `Dockerfile` in `sa-logic` folder:**

```
FROM python:3.6.6-alpine
COPY sa /app
WORKDIR /app
RUN pip3 install -r requirements.txt && \
   python3 -m textblob.download_corpora
EXPOSE 5000
ENTRYPOINT ["python3"]
CMD ["sentiment_analysis.py"]
```

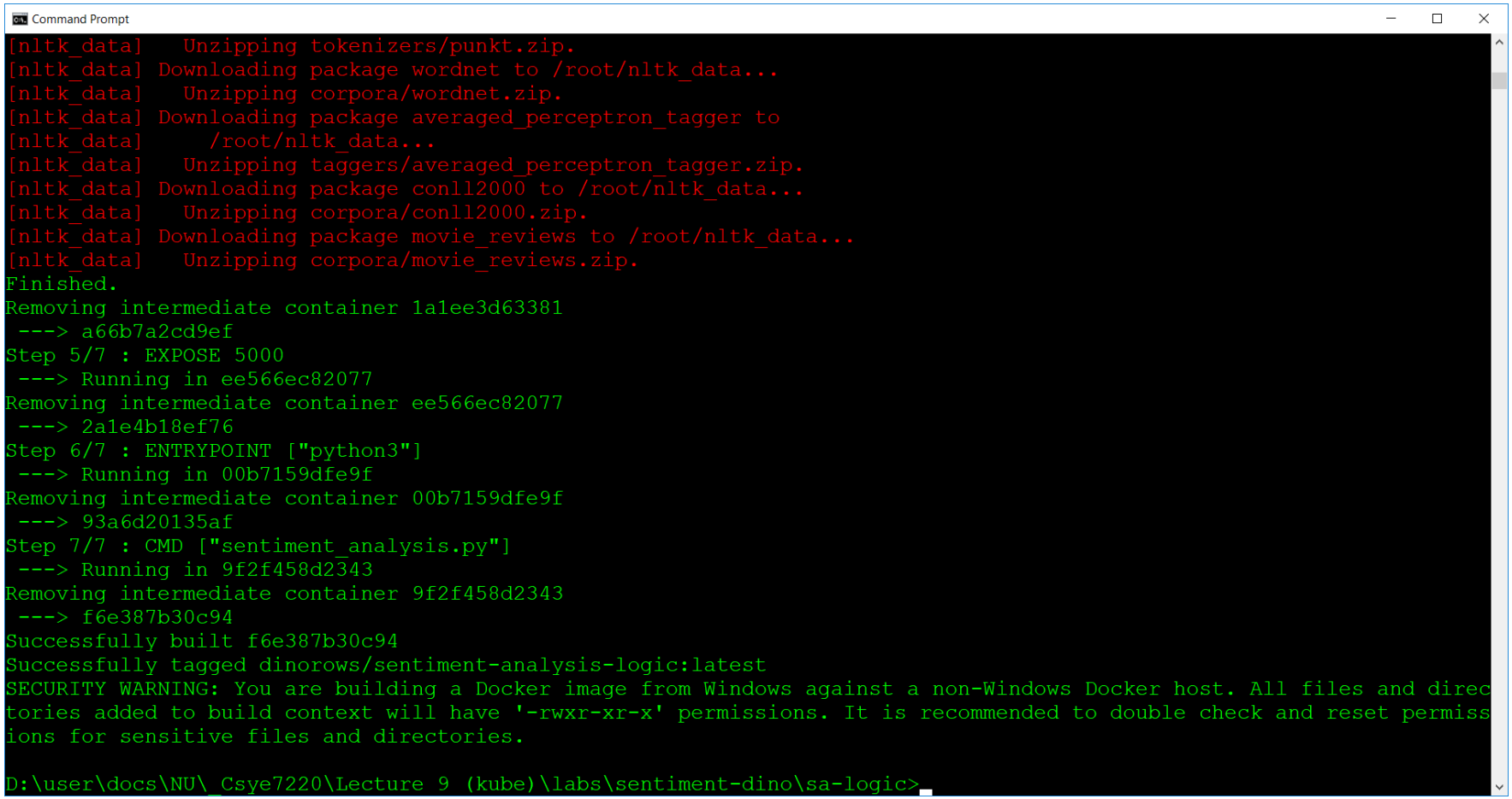
You probably will have to use a different image…

- **We base our container image on alpine linux with a 3.6.6 python engine**
- **We install dependencies**
- **We expose port 5000**
- **We start our python program**
- **Build: `docker build -f Dockerfile -t <your_DOCKER_USER_ID>/sentiment-analysis-logic .`**
- **Push: `docker push <your-DOCKER_USER_ID>/sentiment-analysis-logic`**

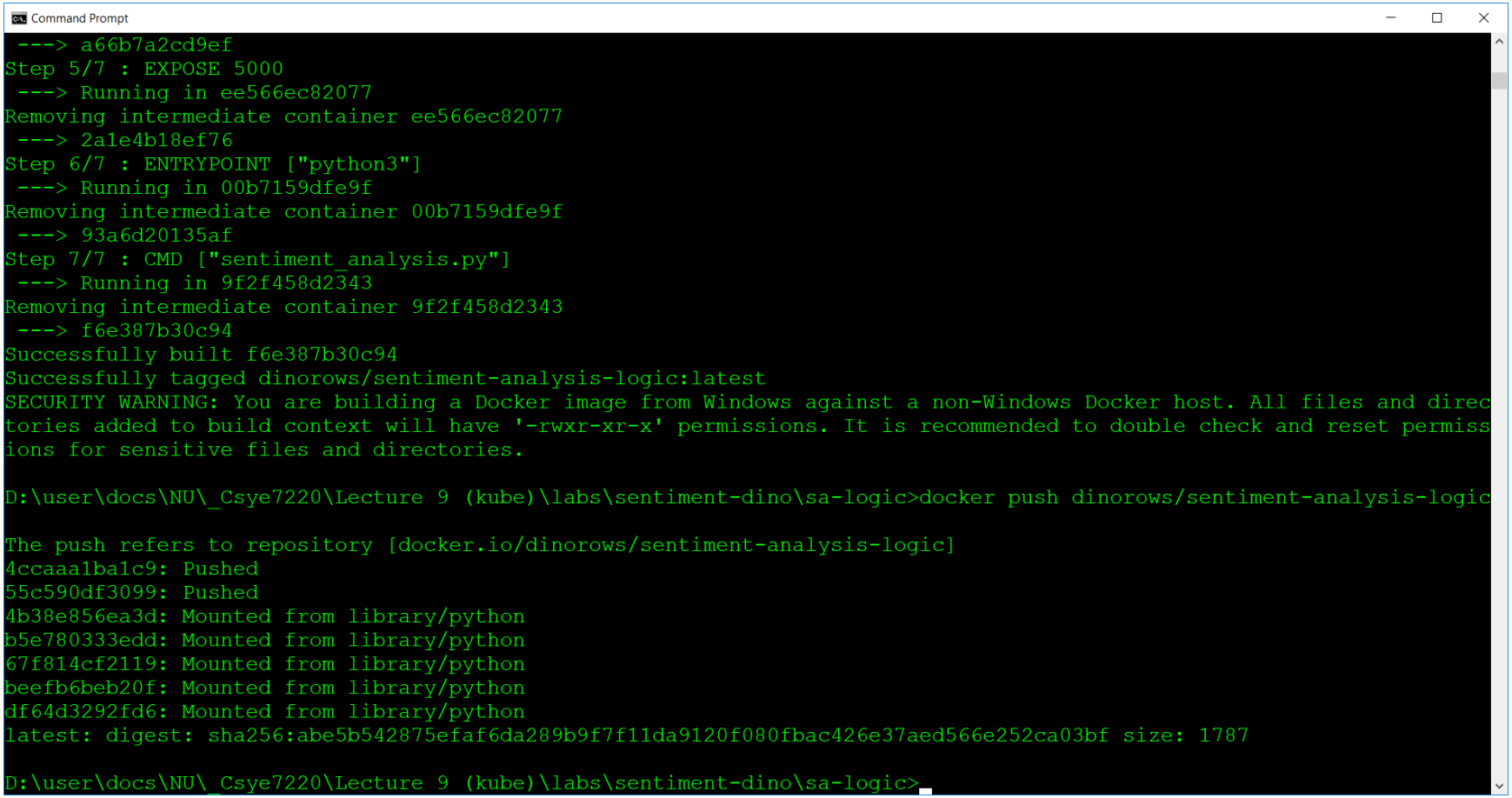
Update!

- Replace docker image
  - `python:3.6.6-alpine`
- With docker image:
  - `python:3.8-slim`
- If you still get gcc errors, then use *this* docker image:
  - `python:rc`

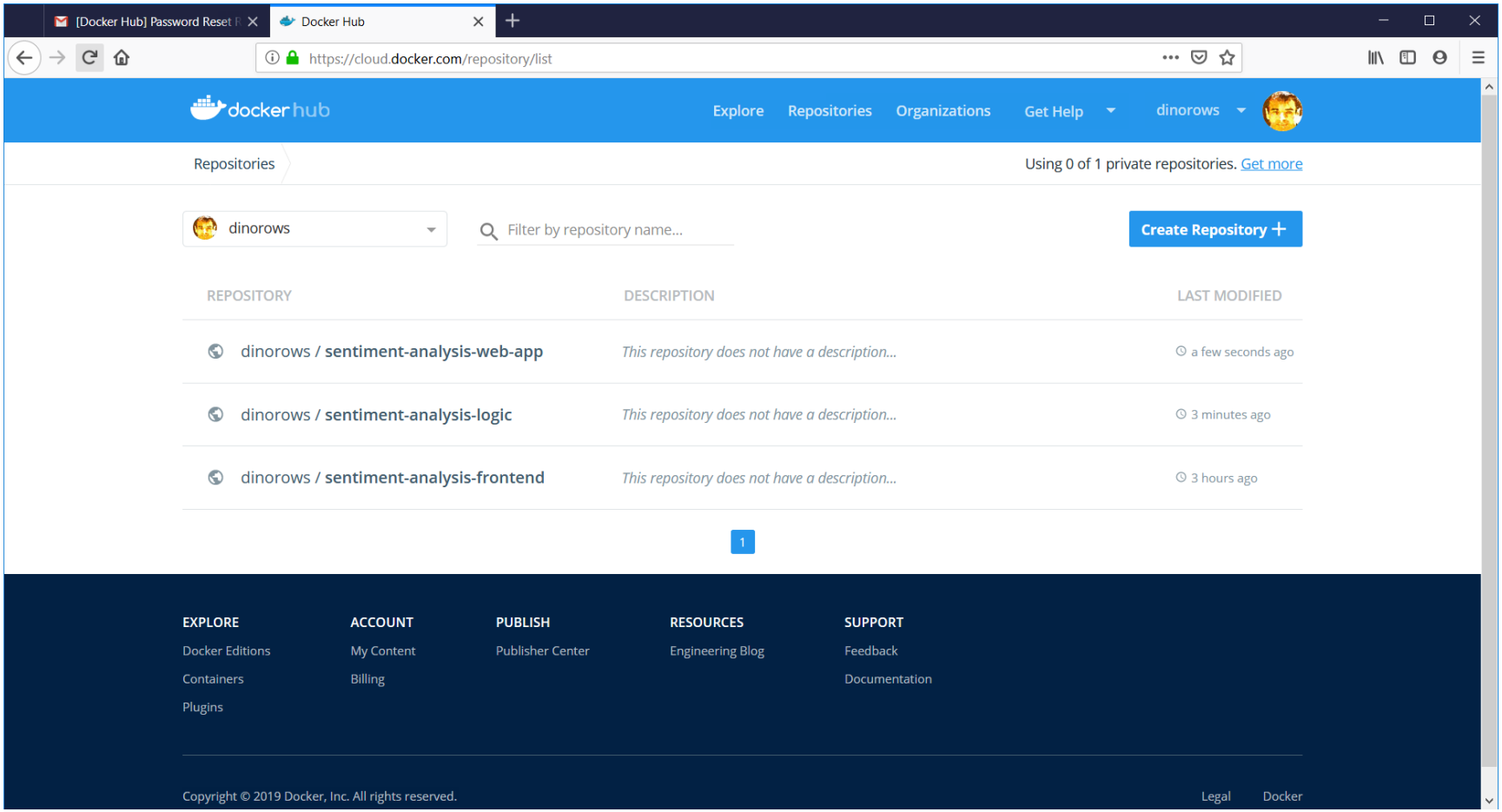
docker build



docker push



Your docker hub, *now*



Important Note

- If you're *not* running docker toolbox, but ***instead you are running Docker Desktop, from Windows or the Mac***, then you are *not accessing the docker-machine IP* and you're using native containers (*not containers running on another VM*), in which case *all your URLs should point to localhost*
  - That means ***you need to modify my source code***, because I am using `docker-machine`, whose URL is 192.168.99.106

How to get the Container IP

- Native docker support needs the Container IP
- If running *native* containers (*not* on VM separate from host machine):
  - Execute:
    - `docker container list`
  - Copy the id of `sa-logic` container and execute:
    - `docker inspect <container_id>`
  - The Container's IP address is found under the property `NetworkSettings.IPAddress`
  - Use it in the `docker run` command
- If using Docker Machine on a VM (e.g using *docker toolbox*):
  - Get Docker Machine IP by executing:
    - `docker-machine ip`
  - *Use that IP in the `docker run` command*

A note about Container environment variables

- You can set any environment variable in the container by using one or more **`-e`** flags, even overriding those already defined with a **`Dockerfile`**
- If the you name an environment variable without specifying a value, then the current value of the named variable is propagated into the container's environment
- **`docker run -e "deep=purple" -e today --rm alpine env`**

A note about Container cleanup

- Clean up (`--rm`)
- By default a container's file system persists even after the container exits
  - This makes debugging a lot easier (since you can inspect the final state) and you retain all your data by default
  - But if you are running short-term foreground processes, these container file systems can really *pile up*
- If instead you'd like Docker to automatically clean up the container and remove the file system when the container exits, add the --rm flag:

Verify first

- `docker container ls`
  - Verify you have *no* other containers running (I don't want other container getting in the way by using ports we're going to use) other than `sentiment-analysis-frontend`

A note on *docker run* commands

- **If you execute docker run in the same folder as your `Dockerfile`, you will start the container, *but also a local instance of your app!***

An important note on *debugging*

- Before starting your container in -d and port mapping mode, run it interactively because this will map app startup errors to your screen!
- So, *before* running this:
  - `docker run -d -p 5050:5000 dinorows/sentiment-analysis-logic`
- Run this:
  - `docker run dinorows/sentiment-analysis-logic`
- *If no errors*, then kill the above container, and run container in –d and –p mode

Secret note for Dino!

- `Requirements.txt` file is missing `flask-cors` dependency!
- See if students can figure this out..

Note: replace `dinorows` *with your docker username!*

All commands on two slides (1/2)

- `docker login -u=dinorows -p=<your-DOCKER-PASSWORD>`
- **sa-logic:**
  - `docker build -t dinorows/sentiment-analysis-logic .`
  - `docker push dinorows/sentiment-analysis-logic`
  - `docker pull dinorows/sentiment-analysis-logic`
  - Run container, listen on port 5050 on host, 5000 in container:
    `docker run -d -p 5050:5000 dinorows/sentiment-analysis-logic`
  - **Other option:** `--rm` **instead of** `-d`
- **sa-webapp:**
  - `mvn install`
  - `docker build -t dinorows/sentiment-analysis-web-app .`
  - `docker push dinorows/sentiment-analysis-web-app`
  - `docker pull dinorows/sentiment-analysis-web-app`
  - Listen on port 8080, change URL + port on which python app listens by overriding the env var SA_LOGIC_API_URL:

```
docker run -d -p 8080:8080 -e
"SA_LOGIC_API_URL=http://192.168.99.106:5050"
dinorows/sentiment-analysis-web-app
```

All commands on two slides (2/2)

- **sa-frontend:**
  - `npm run build`
  - `docker build -t dinorows/sentiment-analysis-frontend .`
  - `docker push dinorows/sentiment-analysis-frontend`
  - `docker pull dinorows/sentiment-analysis-frontend`
  - `docker run -d -p 8085:80 dinorows/sentiment-analysis-frontend`

```
D:\user\docs\NU\_Csye7220.fa19\Lecture 3\labs\sentiment-dino2\sa-logic>docker image ls
REPOSITORY                              TAG                 IMAGE ID            CREATED             SIZE
dinorows/sentiment-analysis-logic       latest              c9d24425500d        49 seconds ago      240MB
dinorows/sentiment-analysis-frontend    latest              cca39f2a0709        10 minutes ago      111MB
dinorows/sentiment-analysis-web-app     latest              292809b1d818        11 minutes ago      125MB
dinorows/sentiment-analysis-logic       <none>              2fa8800837a3        19 minutes ago      237MB
dinorows/georgemichael5000              latest              3f71fd449f46        3 months ago        263MB
```

  - `docker rmi 2fa8800837a3`

All *docker run* commands on one slide

- `docker-machine ip`
- Note: `docker run -p hostport:containerport`
- `docker run -d -p 5050:5000 dinorows/sentiment-analysis-logic`
- `docker run -d -p 8080:8080 -e "SA_LOGIC_API_URL=http://192.168.99.106:5050" dinorows/sentiment-analysis-web-app`
- `docker run -d -p 8085:80 dinorows/sentiment-analysis-frontend`

- Note: can also use `-rm` option instead of `-d`

How to test

- **How to test:**
  - **Run all 3 apps**
  - **Go to python web page and test `/testHealth` endpoint**
  - **Go to springboot web page and test `/testHealth` endpoint**
  - **Go to python web page and test `/testComms` endpoint**
  - **Go to springboot web page and test `/testComms` endpoint**
  - **Go to python web page and test `http://192.168.99.106:5050/analyse?sentence=i+am+so+happy!`**
  - **Go to springboot web page and test `/testSentiment` endpoint**
  - **Finally, go to ReactJS webpage and test a happy/sad sentence!**

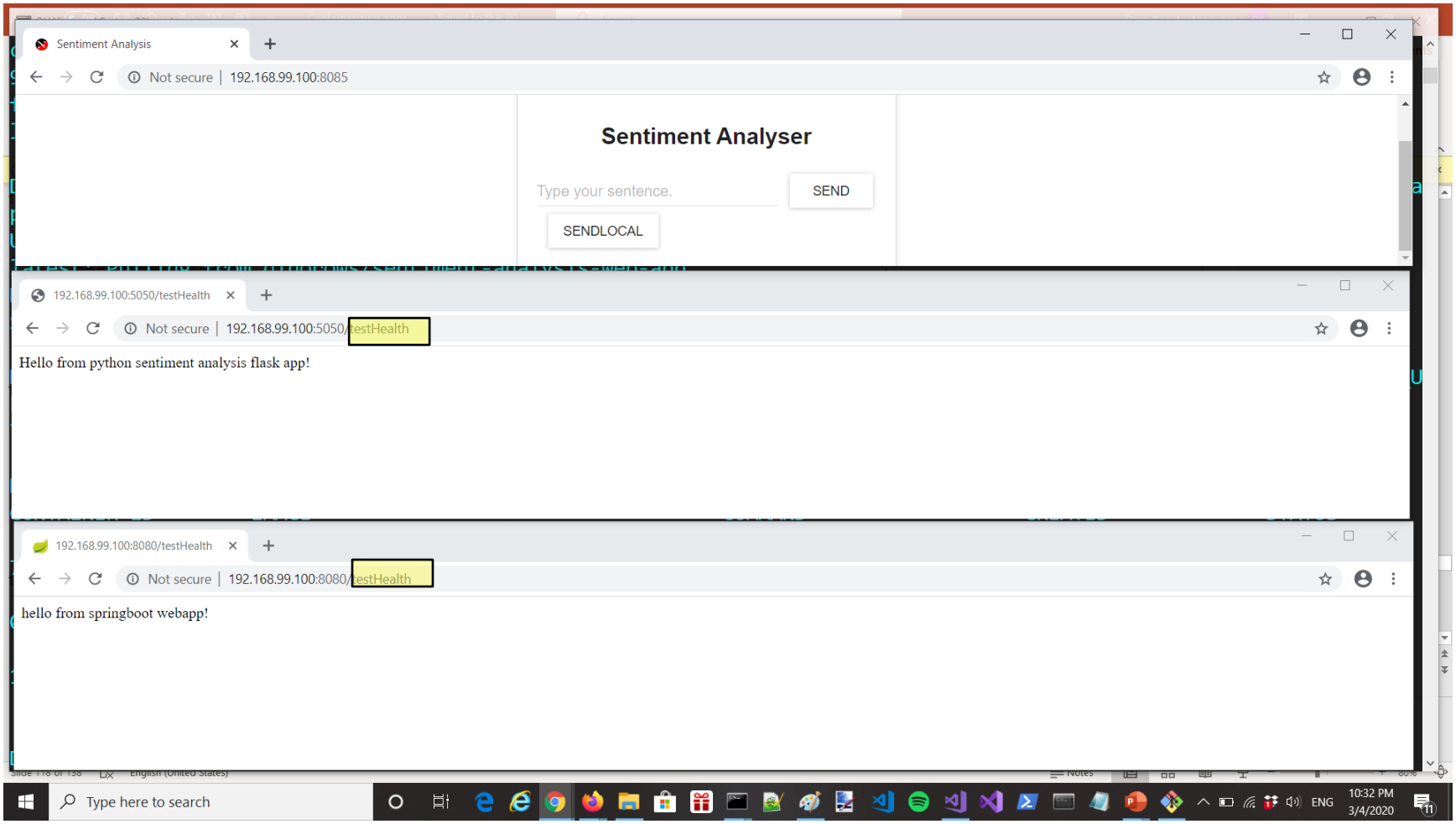
Test container health & inter-container comms


```
D:\user\docs\NU\_Csye7220.fa19\Lecture 3\labs\sentiment-dino\test>docker container ls
CONTAINER ID        IMAGE                                    COMMAND                  CREATED             STATUS
       PORTS                    NAMES
a30a418f3f57        dinorows/sentiment-analysis-web-app      "java -jar sentiment…"   7 seconds ago       Up 5 seconds
       0.0.0.0:8080->8080/tcp   admiring_swartz
33b4eb59eab2        dinorows/sentiment-analysis-frontend     "nginx -g 'daemon of…"   2 minutes ago       Up 2 minutes
       0.0.0.0:8085->80/tcp     loving_liskov
aaf2ff2a42b6        dinorows/sentiment-analysis-logic        "python3 sentiment_a…"   About an hour ago   Up About an h
our    0.0.0.0:5050->5000/tcp   peaceful_lamarr
```

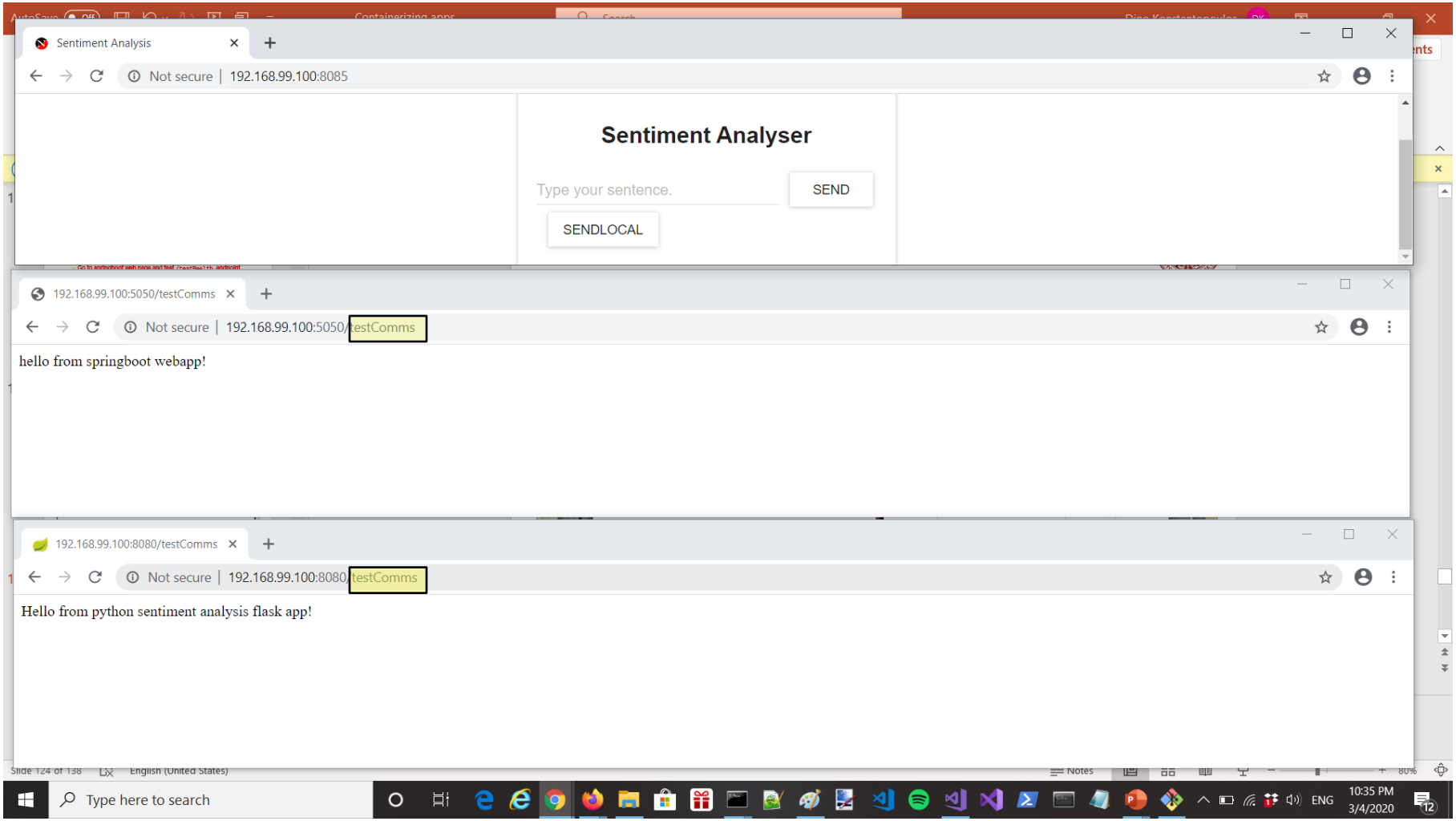
- http://192.168.99.106:5050/testHealth
- http://192.168.99.106:5050/testComms
- http://192.168.99.106:8080/testHealth
- http://192.168.99.106:8080/testComms

- http://192.168.99.106:8085/

/testHealth endpoints (192.168.99.106 host)

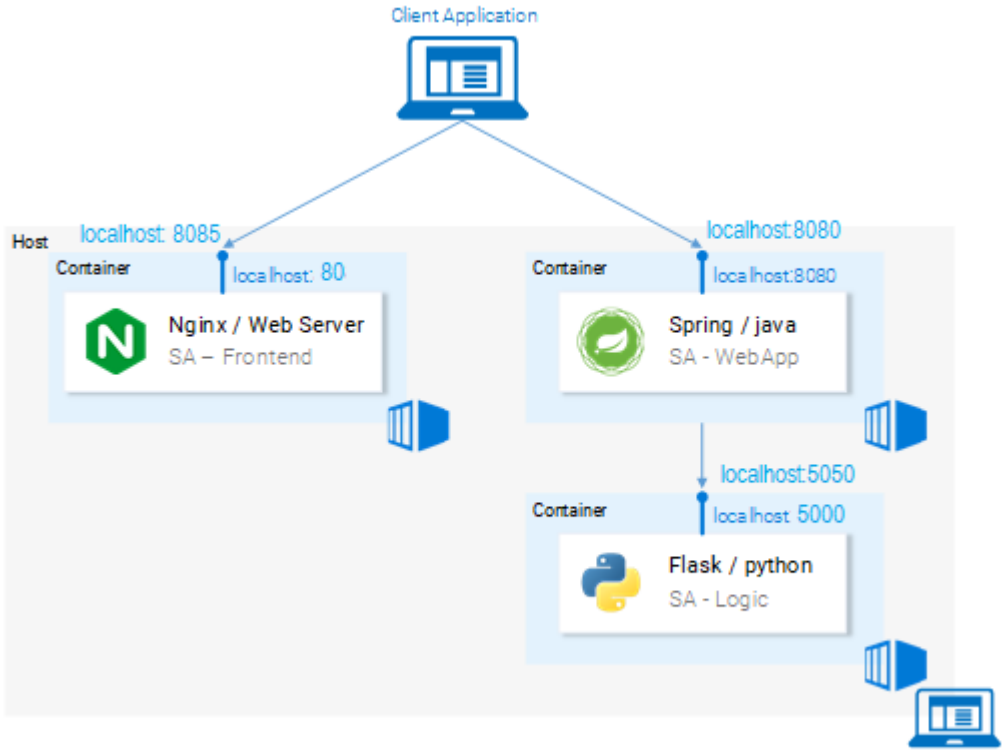


/testComms endpoints (192.168.99.106 host)

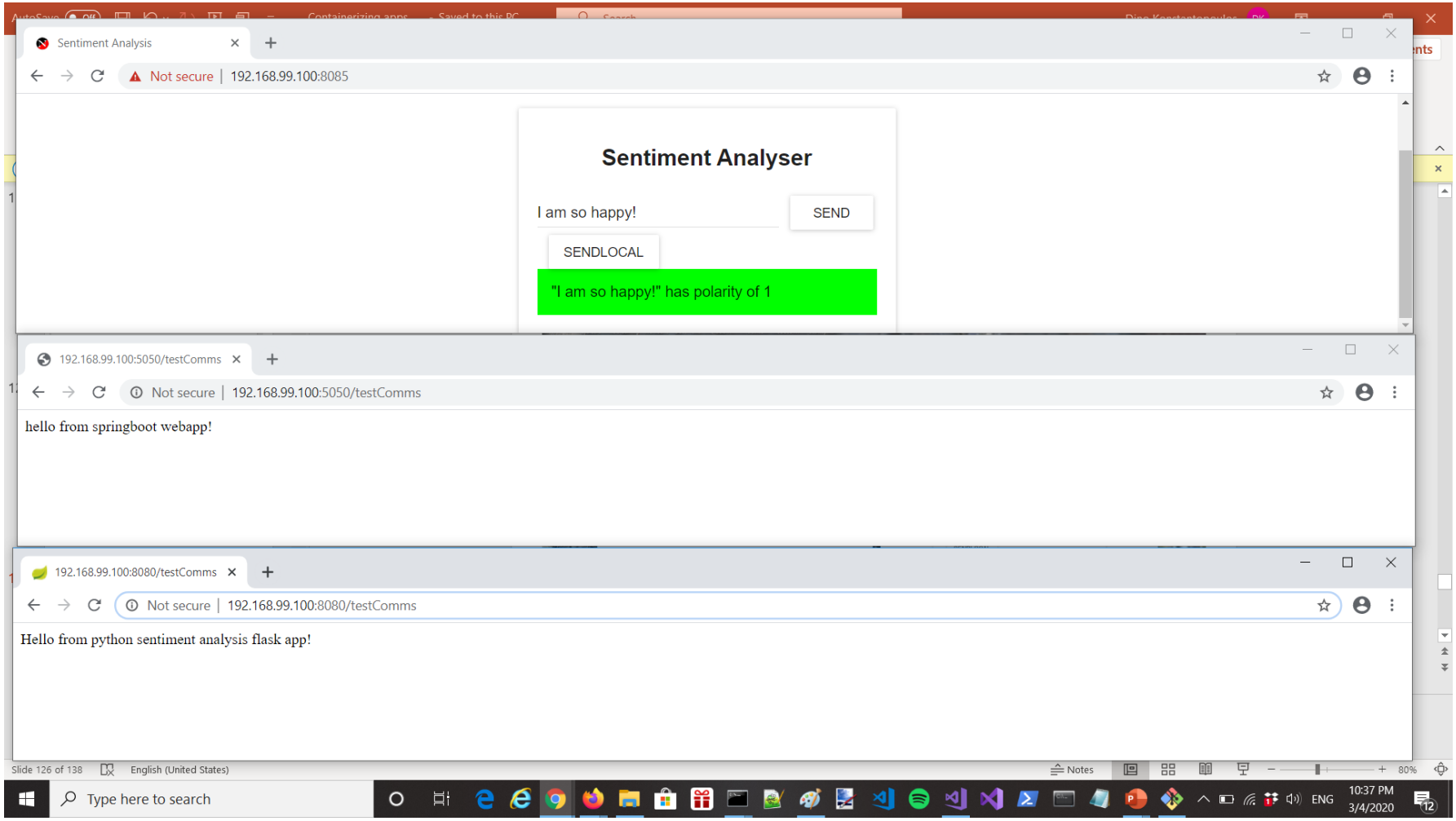


Test full Client/Server app

- Browse to **`frontend`** UI: **<container-ip or docker-machine ip>:8085**
  - **For me:** **`http://192.168.99.106:8085`**



Testing full app (192.168.99.106 host)



Notes

- If using `Docker for Desktop` (Windows or Mac) instead of `Docker Toolbox`, then use `localhost` instead of `192.168.99.106` host (linux VM)
- You may also pull images for this demo from my dockerhub
  - User `dinorows` , *but I would prefer you try your images on your dockerhub*

Creating a network for containers to share

- Communication issues? Create a network!
  - `docker network create --driver bridge sa_network`
  - `docker network inspect sa_network`
- Create a network with a specific subnet and gateway:
  - `docker network create --driver=bridge --subnet=192.168.2.0/24 --gateway=192.168.2.10 sa_network`
- Connect a container to a network
  - `docker run --network= sa_network -itd --name=sa-fe-container sa-fe-image`
  - Any other container you create on this network would be able to automatically connect to one another
- To remove a user-defined bridge network
  - If containers are connected to the network, disconnect them first: `docker network disconnect network sa-fe-container`
  - `docker network rm sa-network`
- https://docs.docker.com/network/bridge/

Debugging containers

- The docker exec command will let you run arbitrary commands inside an existing container
  - `docker exec -it <container_name> bash`
  - `docker exec -it <container_name> sh`
  - `docker exec -it <container_name> /bin/sh`
- To get into the container interactively
  - `docker run -it --entrypoint /bin/bash <container_name>`

Text-based browser in container

- **`docker exec container_id apt-get update`**
- **`docker exec container_id apt-get install -y curl wget`**
- **`docker exec container_id apt-get install -y lynx`**

- **`docker exec container_id sh`**
- **`curl http://localhost:80`**

- **Note: can also do this in `Dockerfile`:**

```
from ubuntu:16.04
run apt-get update
run apt-get install -y curl wget
```

More on debugging containers

- **View stdout history with the logs command**
  - **`docker logs <container_name>`**
  - This history is available even after the container exits, as long as its file system is still present on disk (until it is removed with **docker rm**). The data is stored in a json file buried under /var/lib/docker
- **Stream stdout with the attach command**
  - **`docker attach <container_name>`**
  - By default this command attaches stdin and proxies signals to the remote process. Options are available to control both of these behaviors. To detach from the process use the default `ctrl-p ctrl-q` sequence
- **Execute arbitrary commands with exec**
  - **`docker exec <container_name> cat /var/log/test.log`**
- interactive shell in the container
  - **`docker exec -it <container_name> /bin/sh`**

More on debugging containers

- **Get process stats with the top command**
  - `docker top <container_name>`
- **View container details with the inspect command**
  - `docker inspect <container_name>`
  - Probably the most valuable use of inspect is getting the values of environment vars. For automated deployments, the wrong arg may be passed to a command and a container ends up running with vars set to incorrect values
- **View image layers with the history command**
  - `docker history <container_name>`
  - Shows the individual layers that make up an image, along with the commands that created them, their size on disk, and hashes.

Run *one* process in each container

- Best practice that will make debugging, and reasoning about the state of your container a lot easier
- Docker containers are meant to run a single process that is PID 1 inside the sandbox
- You can make that process a shell and spin up a bunch of stuff in the background, or you can make it supervisor and do the same, but it's not really what containers are good at and it hobbles systems meant to manage and monitor them
- The main benefit of running one process per container is that it's easier to reason about the state of the container at run time
- The container comes up when the process comes up and dies when it dies, and platforms like Docker Compose, kubernetes, and Amazon ECS can see that and restart it
- They can also monitor the health (liveness and readiness) of a single-process container, but it is much more difficult to define declaratively what either of those things means when they depend on multiple processes being in good health

Extreme: Debug a container from another

- Create a debug container with strace
  - ```
    FROM alpine
    RUN apk update && apk add strace
    CMD ["strace", "-p", "1"]
    ```
- Build the container
  - ```
    docker build -t strace .
    ```
- Run strace container in the same pid and network namespace
  - ```
    docker run -t --pid=container:baadf00d \
    --net=container: baadf00d \
    --cap-add sys_admin \
    --cap-add sys_ptrace \
    strace
    ```
  - This attached strace to the baadf00d process and follows it as it executes
    ```

Root filesystem

- To get to the root filesystem of the remote container, use the alpine image and launch a shell, in the same pid and network namespace
  - ```
    docker run -it --pid=container:baadf00d \
    --net=container:baadf00d \
    --cap-add sys_admin \
    alpine sh
    ```
- **With this container attached to the original we can do more debugging**
  - **You can still debug the network but make sure you use localhost because your new sh process is running in the same network namespace**
  - ```
    apk update && apk add curl lsof
    curl localhost:2015
    ```
  - ```
    lsof -i TCP
    ```
  - All standard debugging tools should work from this 2nd container without tainting the original container
    ```

Deleting containers

- `docker stop <container-name>`
- `docker rm <container-name>`

Deleting images

- `docker rmi <image-name>`

The beauty of --rm mode

- `docker run --rm` has the substantial advantage of printing to console all print statements
- `docker run -d` … does not!
  - Need to replace `print()` statements with `sys.stderr.write()` in order to write to container logs!
- This is the best debugging tool ever!
  - It lets you peek at runtime what is going on in your container
- In fact, it's going to be key in debugging all the problems you are going to have with this lab ;-)

And…

- **Remember…**

Beginners

Experts

Homework

- For next week, finish installing docker (DD?), make it work, write a Dockerfile for each one of your sa services, build, run containers and try to make them work together
  - You may have to create a *docker network*
  - You may have to use *Docker Compose*
- Dockerize your rockstar web app (dotnet or react version)

Coming up Lectures Goals

- We'll orchestrate (manage) our Containers with `Kubernetes`!
- We'll learn how to deploy on the Cloud (AWS/Azure) with `terraform`