ECS781P: Cloud Computing
Lab Instructions for Week 4

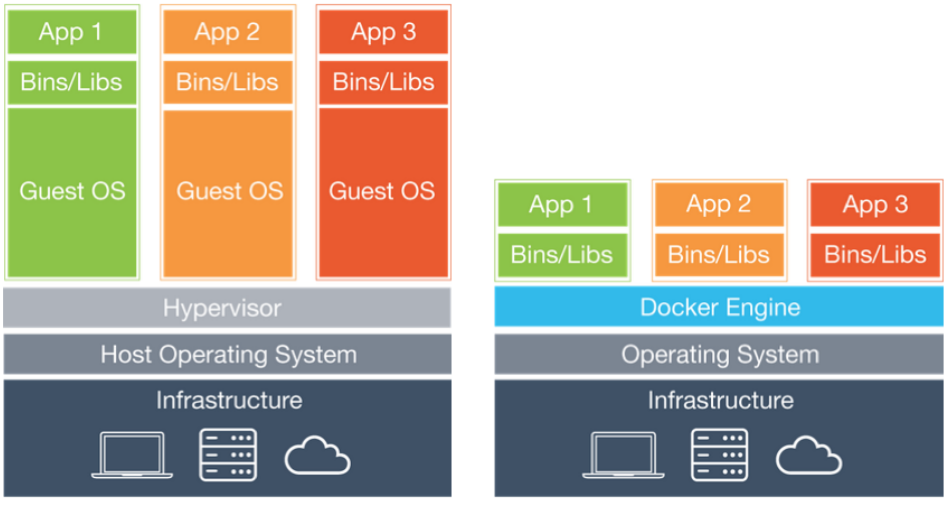
Introduction to Application Containers, and docker

Dr. Arman Khouzani, Dr. Felix Cuadrado

Jan 30, 2018

1 Application Containers

Application containerization, or simply, containerization is closely related to virtual machines, but has very important distinctions. In VM, the entire OS is emulated (including all the functionalities of the kernel) along with all the resources (CPU, RAM, Storage, Networking, I/O). This is good when our purpose is exactly to have a working machine (desktop, or server) that is not tied up to its underlying infrastructure. But if our objective is to run applications that will run on any hardware and OS, then spinning an entire virtual machine looks like a rather waste of the host resources. This is where the idea of containerization becomes interesting.



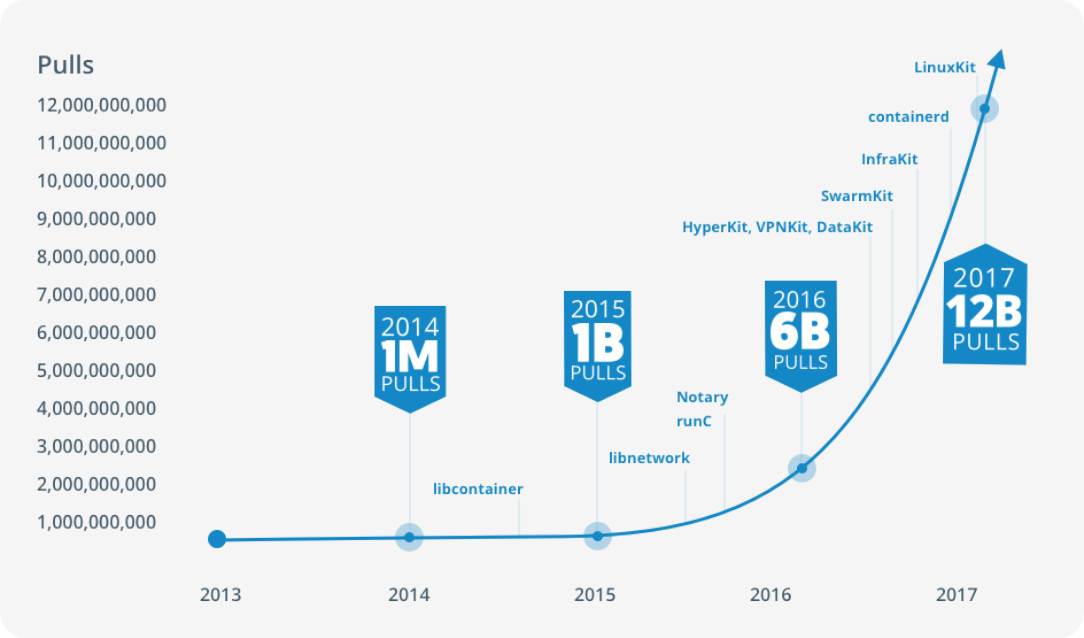


Figure 1: Ref.: `https://www.docker.com/what-container` (Accessed 30/01/2018)

While a virtual machine provides an abstract *infrastructure* (a distinct OS, CPU, RAM, storage, networking, I/O), the container provides an abstract *OS*. In particular, while a VM can run any application compatible with the machine it is emulating, a container of a program contains only a subset of an operating system (programs and libraries) and system resources (CPU, RAM, Storage, etc) that is needed to run that specific program. In particular, the OS **kernel is shared** between the host machine and the containers.[1] This increases the utilization of the underlying resources, allows us to pack more applications onto the same physical resources.
The main company pushing the idea of containers was `Docker`. The graph of number of pulls per year in Figure 1 should give you an idea! Docker took advantage of `LxC` (`LinuX Containers`), which uses the exiting Linux kernel's functionalities and concepts like `cgroups` to create and run multiple isolated Linux virtual environments on a single host, and created a user-friendly interface for managing them.
Due to its popularity and widespread use, we think it is a good idea to add it to our arsenal of practical skills. Before we proceed with the lab manual, you may wish to have a quick read of the first section of the Docker-for-Virtualization-Admin-eBook (pages 3–6: Containers are not VMs, as well as a quick visit to the docker website: `https://www.docker.com/`. Also note that there are good online resources and documentation on docker, including docker's own documentation: `https://docs.docker.com/`.

[1] In fact, this 'shared kernel' is what raises some security issues since the "isolation" is not fundamentally complete and container escaping can be a threat.

2 Preparation

If you are using your own machine, then please install docker ('community edition') `docker-ce` from Install Docker, the version that matches your machine. We have installed it on an `ubuntu-16.04 LTS` image for you, on which, you can log in with `root` privilege.[2]

1. Open a terminal and run `vminstance`.
2. If you have any (difference) images from before, delete them (to free up your disk quota, unless you have some important data on any of them). Just press the down arrow key to get on their name, then press enter, and then select delete. (this is equivalent to just deleting the difference image files from your local image folder at `~\Images`). You should now see the blank menu as in Figure 2.

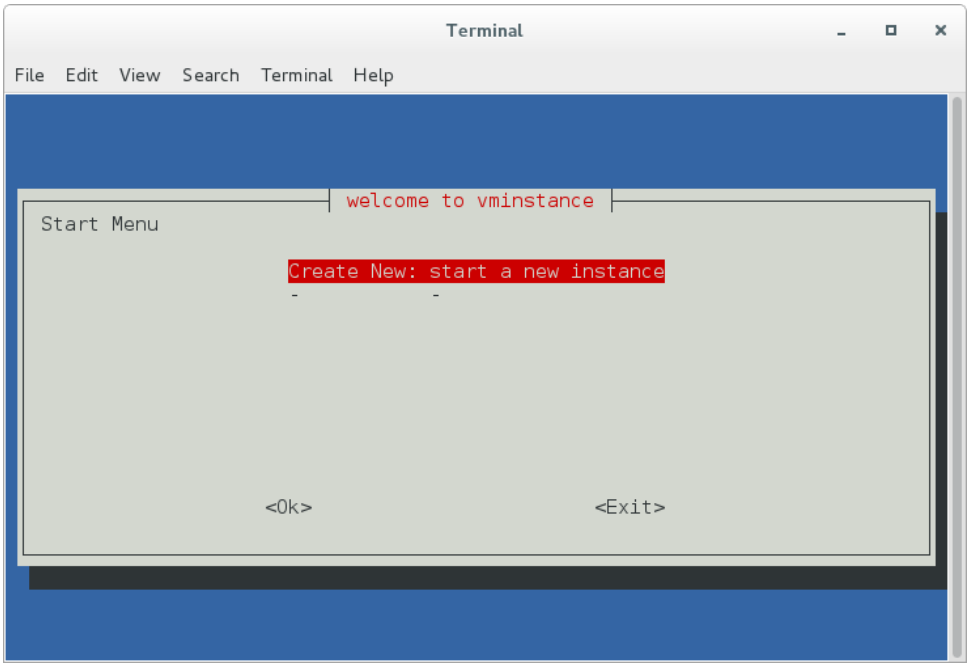


Figure 2: Step 2 of "Preparation".

3. Select `Create New: Start a new instance` (enter). Scroll down (using keyboard) the menu of available images, to select `Ubuntu_16_Docker (Snapshot)`, as in Figure 3.
4. Your VM should start booting! If it did not, then from `vminstane` main menu, you see it, accompanied with the label `RUNNING`. Select that option, and from the subsequent menu, select `Connect: connect to running instance`, as in Figure 4.

[2]Note that if you are installing docker for yourself, you do not necessarily need to do this inside a VM. In fact, it is generally not a good idea to run docker engine (which is a kind of virtualisation technology) inside another virtual machine (because nested virtualisation can have a high overhead and low efficiency. However, for security/technicality considerations, we will be inside a VM! For a good discussion on pros and cons of this, read the third section of Docker-for-Virtualization-Admin-eBook (pages 8–9: Physical or Virtual?)

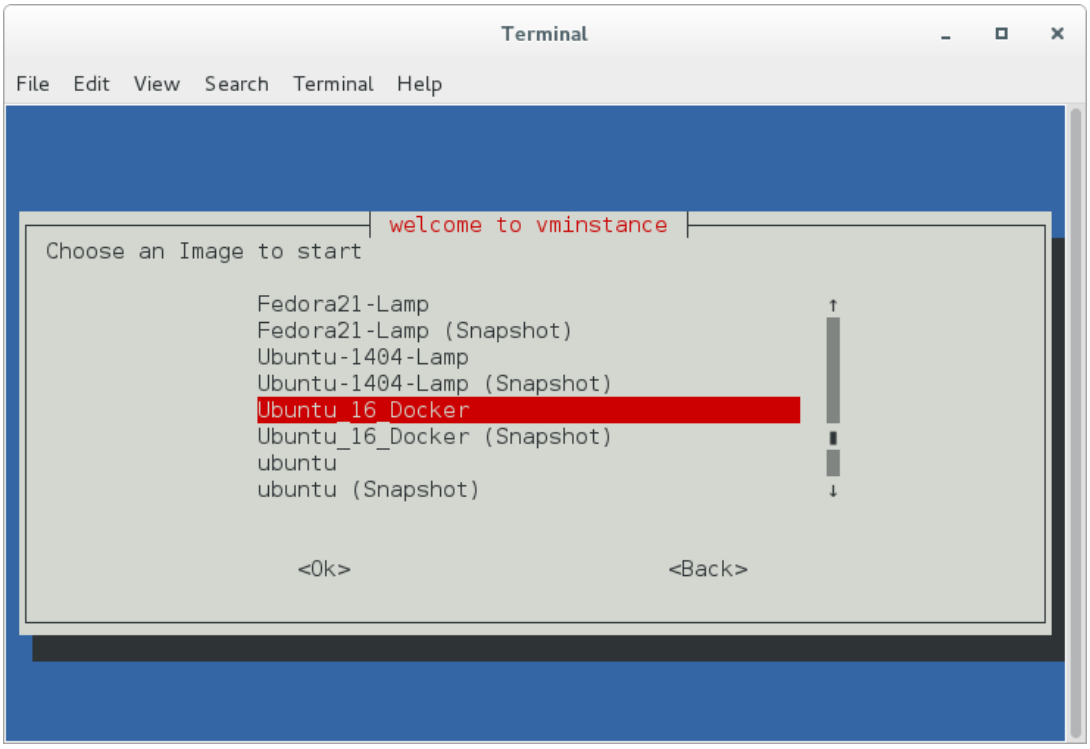


Figure 3: Step 3 of "Preparation".

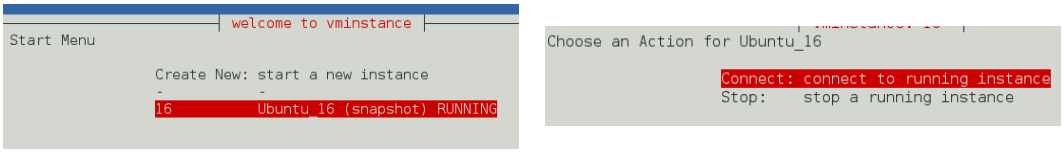
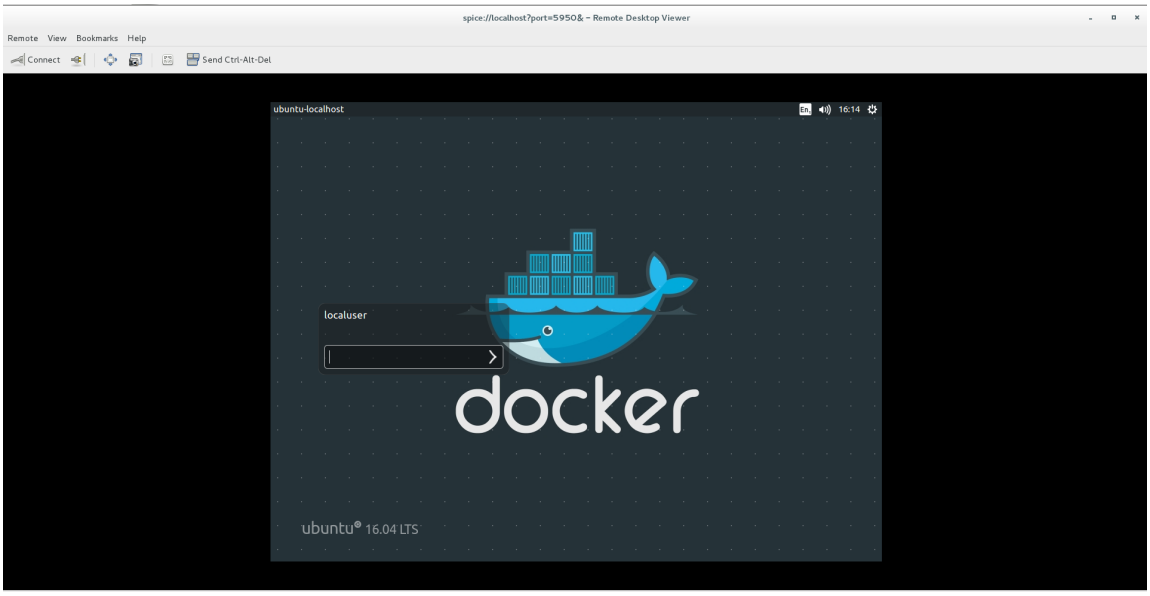


Figure 4: Step 4 of "Preparation".

5. When prompted to provide a login password, it is `localuser` (the same as your username) as seen below. (Also later on, when you run commands with root privilege using `sudo`, if you get prompted for the root password, it is `localuser`).



3 Docker 101

The most important concept is the difference between a "container" (or a container instance) and an "image" (or a container image): To use a non-technical analogy, an **image** for an application is like the recipe and essentials to build a self-sufficient environment for that application from. So the images are NOT the encapsulated environment itself. The image can be used to create as many actual instances of the encapsulated environments as desired. Each of these actual instances created from an image is a **container**. Now to use an object-oriented-programming analogy: you can think of an "image" as a *class*, which can be used to create "containers" as *objects instances* of that class. Indeed, we had the same concept in the context of virtual machines as well: an image of a virtual machine is like the class, from which as many VM "instances" can be created (destroyed, then re-created, etc.) We will see that the analogy goes further: we can in fact create our own images by layering base images on top of each other, similar to when we "extend" a class!
The program that provides the virtualization, creating container instances from images, running and managing them, is called the *docker engine*. It is composed of a *docker daemon*[3] `dockerd` and a friendly application interface (as a command line interface) called *docker cli*, which is invoked simply by `docker` command. People may just say *docker* to refer to either one, and which one they actually mean should be understood from the context. It is worth reviewing these terminologies and concepts clear at least once. So as your first exercise, please visit the following page: Docker-Overview, and provide a brief description of the following terminologies and their respective role:

Docker Engine, Docker daemon, Docker client:

Docker registries:

Docker images, containers, services:

The hello-world image

1. Open a `terminal` on the `ubuntu_16_docker` VM we logged into.
2. Issue `docker`, you should see a summary list of available options and arguments.
3. As you can guess, there is a `hello-world` image! You can use it to check whether our docker installation is working properly:

   ```
   $ sudo docker run hello-world
   ```

   This will first look for the image `hello-world` locally, if it cannot find it, then tries to search for it in the default docker registry online, finds it, downloads it, and runs it in the foreground. This particular image immediately terminates after prompting a message. The message is very important. Please read it carefully!

[3] Daemons are just called to long-running processes in Linux.

Basic operations: search, pull, images, run, ps, inspect, pause/stop/remove

Before we get to some interesting exercises with Docker, we need to get comfortable with some basic docker commands:

1. **`search:`** All existing docker images at https://hub.docker.com/explore/ can be found by using the command `docker search <image-name>`. For example, to find an image for `nginx`, you would use:

   ```
   $ sudo docker search nginx
   ```

   What would be the command to search for an image of `redis`? what about `ubuntu`?

2. **`pull:`** Once you find an image, it can be downloaded using `docker pull <image-name>`. Pull the `nginx` image. Note: using the default name pulls the `latest version` by default. If we want a specific version (which is always a much preferred approach) we should explicitly specify it: `sudo docker pull nginx:x.y`.

3. **`images:`** to see a list of locally available images (images pulled onto our host machine), you can issue `docker images`. See the list of pulled images, you should see two of them. What are they? There are multiple columns, showing the attributes of the image. What each represent?

4. **`run:`** to create container instances from an image, we use
   `docker run <options> <image-name>`.
   In the most simplest form: `docker run -d <image-name>`: the `-d` command runs the container as a background (isolated/sand-boxed) process. This is usually how we want the containers to be running anyway (in the background).

5. **`ps:`** to see the list of running containers and their main attributes, you can use `docker ps`. Issue this, and analyze what you see. What is each column representing? Specially, what is the funny name in the last column? Also, what do optional arguments like `docker ps -a` and `docker ps -aq` do?

   - Although we logically first pull the image and then ran it, the process of downloading can be done automatically by `docker run` if it does not find the image locally. For instance, try the following:

     ```
     $ sudo docker run -d redis
     ```

     Issue this, and then use `docker ps` to see if the container is created and is running.

6. **`inspect:`** To *inspect* a container, we can use `docker inspect <container-ID/name>`. Use it to gather information on our containers, and try to parse the output as much as you can.

7. **`stop/rm/rmi`** Find out what is the difference between the commands `docker stop <name>`, `docker rm <name>` and `docker rmi <name>`. (Hint: remember that containers are like instances of a class).

8. To find out what port number a container is running on, we can issue: `docker port <container-` Find out the port number of our containers. Can you tell why these ports were selected?

9. The containers are completely isolated and sand-boxed. So in order to communicate with them, we need to expose their ports. In particular, we will bind our host ports to those of the container. For instance, we can bind the port of the nginx container to our host machine's port using the following:

```
$ docker run -d -p 8080:80 nginx
```

   - Question: is 8080 in our host machine or the container?
   - Question 2: how about 80?
   - Question 3: What is the ip address on our host machine on which this port number is used for binding? (Hint: you can use `docker ps` to gather such information.
   - Question 4: How can we bind a specific port on a specific ip address of the host with the container?

3.1 Building our own container images, hosting a simple static HTML page over nginx

As our first step beyond *hello-world*, we will run a *hello-world* static html page using docker! It will be served using the popular `Nginx` (pronounced "engine-x") as our web-server, but inside a container! Nginx "is an open source reverse proxy server for HTTP, HTTPS, SMTP, POP3, and IMAP protocols, as well as a load balancer, HTTP cache, and a web server (origin server)." https://hub.docker.com/_/nginx/. In particular, we use its `alpine` variation (image name `nginx:alpine`), which is based on the ultra-light-weight "alpine" Linux.
For this, we need to add a few things to the plain `nginx` image. For one thing, our static `index.html` page! We do this, by creating our own image, starting from the base image of `nginx`. The way this is done is by using a specific text-edited file called, not so imaginatively, the `Dockerfile` (no extension needed).

1. Create your own static html file in a directory, using your favorite text-editor, for instance, `nano`:

   ```
   sudo nano index.html
   ```

   Here is mine:

   ```
   <h1> Hello to Jason Isaacs! (from inside a container!) </h1>
   ```

   Save it and exit.

2. Now, in the same directory, create our first `Dockerfile`:

   ```
   FROM nginx:alpine
   COPY index.html /usr/share/nginx/html
   ```

Again, save and exit. This needs a bit of explanation:

- the keyword `FROM` specifies the base image we are creating our image from (similar to which class we are extending!). For our purpose, we use `nginx:alpine`.
- the keyword `COPY` effectively does what it says: it copies our files (here, our static html page) into a directory inside the container image! As you may know, nginx serves static pages from the default directory of `/usr/share/nginx/html`. So that where we are copying our file to.

3. The Dockerfile is the "recipe" to building a new image. We then need to actually building the new image by using, (can you guess!), `docker build`:

```
$ sudo docker build -t hello-world-webserver-image:v1 .
```

   The only mystery part is the argument `-t`: we use it to `tag` our image, and we have gone ahead and added a `v1`, to indicate this is version 1 of this image!

4. You can check that this is now added to your list of available images by issuing `docker images`.

5. Now, we are all ready to run our container and test it:

```
$ sudo docker run -d -p 80:80 hello-world-webserver-image:v1
```

   Check on your container by issuing `docker ps`.

6. To check your success, open a web-browser, and visit `0.0.0.0`. What do you see? If you indeed see your static page, that means you have successfully created and ran your very first application container!