**PuTTY is a terminal emulator for Windows. How to download and install it:**

PuTTY is an open source terminal emulator for Windows. It's pretty much the defacto way to connect from Windows to Linux. This guide will walk you through how to install PuTTY on your Windows PC.

Go to the PuTTY website: [https://www.putty.org/Links to an external site.](https://www.putty.org/)

In the section Download PuTTY, click the **here** link which takes you to the following website:

[https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.htmlLinks to an external site.](https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html)

**Package files => MSI ('Windows Installer') => 64-bit x86 => click on putty-64bit-xxx-installer.msi to download => click on the download to run the installer**

After installing, click on the Windows system menu icon and verify that there is a putty group called **PuTTY (64-bit)**. Make sure you can run the following executables:

* PuTTY
* PyTTYgen

**WinSCP is a file transfer and file editing program for Windows. How to download and install it:**

WinSCP is an open source file transfer program for Windows. It uses the same foundational layers as PuTTY. It's pretty much the defacto way to transfer files between Windows and Linux. This guide will walk you through how to install WinSCP on your Windows PC.

Website: [https://winscp.netLinks to an external site.](https://winscp.net/)

Click on the big green DOWNLOAD NOW button.

Scroll down and click on the big green DOWNLOAD WINSCP xxxx (xxx MB).

Click on the downloaded file to run the installer.

It should place an icon on your desktop and also an icon in the Windows system menu (lower left Widndows icon).

**Private keys in pem format (used by AWS) need to be converted to PuTTY Key format, which is used for both PuTTY and WinSCP:**

PuTTY (and also WinSCP) store private keys in a different file format than pem. The pem file create for your private key by AWS will need to be converted. This guide walks you through the conversion.

The PuTTYgen app will be used for the conversion, start it:

**Windows system menu (lower left) => PuTTY (64-bit) => PuTTYgen**

Click the Load button (lower right)

Change dropdown in the lower right from "PuTTY Private Key Files (\*.ppk) to All Files (*.*)

Your pem file should now show up. Click on the pem file. Click the Open button.

You get a popup saying that the key loaded correctly (hopefully!). Click OK to confirm.

Click on the "Save private key" button in the lower right. It will ask you if you are sure, click Yes.

I usually use the same name as the pem file name as it will have the ppk extension and not overwrite the pem file.

Now you have a ppk file you can use for PuTTY and WinSCP.

Close out PuTTYgen.

**Login to Linux from Windows using PuTTY:**

In AWS, get the Public DNS for your instance:

**AWS => N. Virginia => Services => Compute => EC2 => Instances => Instances => select instance => Connect => SSH Client => 4 click on copy icon to copy public DNS**

Start PuTTY: Windows system menu => PuTTY (64-bit) => PuTTY

Host Name: w205@xxxxx where xxxxx is your public DNS

Left panel => SSH => Auth => Private key click the Browse button and find your private key

Left panel => Session (very top you may have to scroll)

Saved Sessions: UCB\_MIDS\_w205

Save

Exit PuTTY => start PuTTY => UCB\_MIDS\_w205 => Load => Open

Should now be logged into your VM !

**Connect to Linux from Windows for file transfers and editing using WinSCP:**

In AWS, get the Public DNS for your instance:

**AWS => N. Virginia => Services => Compute => EC2 => Instances => Instances => select instance => Connect => SSH Client => 4 click on copy icon to copy public DNS**

Start WinSCP: Windows system menu => WinSCP

Host Name: xxxxx where xxxxx is your public DNS

Username: w205

Advanced => Advanced => left panel => Authentication => Private Key File => choose you PuTTY key

Save

Login

Drag and drop files back and forth

**Make Linux level backups (above we made VM level backups - we need both to be safe!):**

# How to Guide: Linux - Running a Backup

Video: [https://kevincrook.com/ucb/mids/w205/how\_to\_guides/linux/08\_linux\_backup.mp4Links to an external site.](https://kevincrook.com/ucb/mids/w205/how_to_guides/linux/08_linux_backup.mp4)

## Backup Scripts

There is an alias created to run backups. To see aliases:

alias

alias | grep backup

To see the scripts:

cd /home/w205/scripts

ls -l

To see the backups after they have run:

cd /home/w205/backups

ls -l

cd /home/w205/backups/full

ls -l

cd /home/w205/backups/partial

ls -l

## Full Backup

It is recommended that you make a full backup once a week and download it to your laptop or desktop.

Full backups will be around 1 GiB, so weekly backups will start to add up on using disk space in your VM. You may only want to keep 3 or so full backups and delete the oldest one when you get to 4. Since you have copied it to your laptop, you have a safe copy elsewhere.

To run a full backup:

backup\_full

## Partial Backup

Partial backups are automatically made every time you start your VM. If you leave your VM running overnight, it will automatically make a partial backup at 2am Pacific time.

It is recommended that you periodically download the partial backups to your laptop or desktop.

Partial backups are very small. They do not backup the data files that are used to load the databases and the actual database files, all of which are large. Since they are so small, you can run them and keep them around without deleting them.

When you finish a big piece of work, you should run a partial backup. For example, say you have been working on your project for 3 hours and are at a stopping point, you should run a partial backup to be safe from losing your work.

To run a partial backup:

backup\_partial

**The VM is set to Pacific time zone. If you want to set it to another time zone, here's how:**

Amazon Linux puts everything in UTC with no local time zone set. In the w205 AMI we set the local time zone to US Pacific. If you are in another time zone and want to change your time zone, this will show you how.

See the current date and time:

date

See the current time zone:

timedatectl

Timezones use the tz database (in my humble opinion the most bizzare list of timezones ever created!):

[https://en.wikipedia.org/wiki/List\_of\_tz\_database\_time\_zonesLinks to an external site.](https://en.wikipedia.org/wiki/List_of_tz_database_time_zones)

See a list of time zones:

timedatectl list-timezones

Set a new timezone:

sudo timedatectl set-timezone America/Chicago

Set the timezone back to Pacific:

sudo timedatectl set-timezone America/Los\_Angeles

**Neo4j - Adding Plugins, Using File URLs, Deleting Entire Database, Increasing Memory**

**"Neo4j" is a graph database management system developed by Neo4j, Inc. It's an open-source NoSQL database that utilizes graph structures for semantic queries with nodes, edges, and properties to represent and store data.**

**Adding Plugins to Neo4j**

Plugins are additional modules that are available in Neo4j.  For example, we are using the Graph Data Science (graph-data-science) plugin.  Other plugins include: APOC (apoc), APOC Core (apoc-core), Bloom (bloom), Streams (streams), and Neo Semantics (n10s).

Plugins are added at the cluster level, that is, they are added to the docker-compose.yml file for the neo4j container specification.

**Step 1: Stop any running Docker containers or clusters**

**Step 2: Move to the directory that contains the cluster you want to add plugins**

In most cases, this will be the anaconda\_postgres\_neo4j cluster

cd  ~/docker/clusters/anaconda\_postgres\_neo4j

**Step 3: Edit the docker-compose.yml file to add any additional plugins**

The neo4j entry should look similar to this (there may be some differences as time passes that we add):

neo4j:

image: w205\_neo4j

volumes:

- /home/w205/user:/user

- /home/w205/docker/mounts/neo4j/data:/data

environment:

NEO4J\_PLUGINS: '["graph-data-science"]'

NEO4J\_AUTH: 'neo4j/ucb\_mids\_w205'

expose:

- "7473"

- "7474"

- "7687"

ports:

- "7473:7473"

- "7474:7474"

- "7687:7687"

Change the line:

NEO4J\_PLUGINS: '["graph-data-science"]'

to add any any plugins you want to add.  For example if you wanted to add APOC, you would change to this:

NEO4J\_PLUGINS: '["apoc", "graph-data-science"]'

The first paragraph gives a list of the strings needed for each module.  Neo4j may add some new modules not in the list as time passes.

**Step 4: Start the Docker cluster and verify that the cluster comes up, and that the Neo4j container comes up without exiting**

**Using File URLs with Neo4j**

Some Neo4j Cypher functions, procedures, etc. may try to read files from the local file system.

Neo4j requires a specific location for the root of the file URL:

/var/lib/neo4j/import/

For example if we use the following file URL:

file:///my\_file.csv

Neo4j will look for the following file in the container:

/var/lib/neo4j/import/my\_file.csv

We will need to mount a directory to the Neo4j container.

However, Neo4j will overwrite the owner, group, and blow away permissions, so we will need to create a special directory just for Neo4j file URLs.

**Step 1: Stop any running Docker containers or clusters**

**Step 2: Create a special directory to hold the Neo4j file URLs**

mkdir ~/neo4j\_import

**Step 3: Move to the directory that contains the cluster you want to add the file URL volume mount to**

In most cases, this will be the anaconda\_postgres\_neo4j cluster

cd  ~/docker/clusters/anaconda\_postgres\_neo4j

**Step 4: Edit the docker-compose.yml file to add a volume mount**

The neo4j entry should look similar to this (there may be some differences as time passes that we add):

neo4j:

image: w205\_neo4j

volumes:

- /home/w205/user:/user

- /home/w205/docker/mounts/neo4j/data:/data

environment:

NEO4J\_PLUGINS: '["graph-data-science"]'

NEO4J\_AUTH: 'neo4j/ucb\_mids\_w205'

expose:

- "7473"

- "7474"

- "7687"

ports:

- "7473:7473"

- "7474:7474"

- "7687:7687"

In the volumes section, add the following volume mount:

- /home/w205/neo4j\_import:/var/lib/neo4j/import

**Step 5: Start the Docker cluster and verify that the cluster comes up, and that the Neo4j container comes up without exiting**

**Step 6: Copy files into the ~/neo4j\_import directory**

It will change the ownership and blow away file permissions, so you will need to use **sudo** to copy files into the directory

Here is an example of copying a file named my\_file.csv in the current directory to the ~/neo4j\_import directory

sudo cp my\_file.csv ~/neo4j\_import

**Deleting the entire Neo4j database files at the Linux level**

There are two main reasons you may want to delete the entire Neo4j database files at the Linux level:

* You suspect your database has become corrupt
* You created a lot of nodes and/or relationships that you run out of memory when you run the function to  wipe out the database by finding and deleting all of them

**Step 1: Stop any running Docker containers or clusters**

**Step 2: Move to the directory that contains the data mount for Neo4j**

cd ~/docker/mounts/neo4j

**Step 3: Rename the data directory**

Renaming it instead of deleting it is a bit safer.  If we need to go back to the old database we can.

Take a quick look to see the data directory:

ls -l

Typically, I will choose today's date for the renamed directory.

sudo mv data data\_2022\_11\_18

Take a quick look to see that the data directory has been renamed

ls -l

**Step 4: Start the Docker cluster and verify that the cluster comes up, that the Neo4j container comes up without exiting**

**Step 5: Verify that a new data directory has been created**

cd ~/docker/mounts/neo4j  
  
ls -l

**Step 6: Run a lab Jupyter Notebook to verify that you can create a graph and run some of the basic algorithms**

**Step 7: Run the Neo4j web based GUI and verify you can query the nodes and relationships**

**Increasing Memory for Neo4j**

Neo4j stores the databases on disk with caching to memory, so you can have very large databases with a small amount of memory.

Basic Cypher queries don't use a lot of memory.

Procedures, functions, etc. in the plugins (such as the Graph Data Science plugin) can use a lot of memory if they are run on large graphs.

For the default scenario, and for the alternative scenarios where we have provided a dataset, so far they have not needed to increase memory.

If you are using another dataset that is really large, and you are running a procedure on it, and run out of memory, you may need to add memory.

Adding memory requires a change at the Docker image level.

**Step 1: Stop any running Docker containers or clusters**

**Step 2: Move to the directory that contains the Neo4j image**

cd ~/docker/images/w205\_neo4j

**Step 3: Edit the Neo4j configuration file neo4j.conf and change the upper limit for memory (heap size)**

The section "Memory Settings" will have numerous settings for memory.

Typically increasing the maximum heap size will do the trick.

Find the following line:

#dbms.memory.heap.max\_size=512m

Most settings in the file are commented out and show the default value.

Uncomment the line by removing the # at the left.

Try moving it from 512m to 1g as follows:

dbms.memory.heap.max\_size=1g

**Step 4: Rebuild the Neo4j image**

docker build -t w205\_neo4j .

**Step 5: Start the Docker cluster and verify that the cluster comes up, and that the Neo4j container comes up without exiting**

**Step 6: You may need to repeat the above steps several times, adding some more memory each time, at some point, you may need to increase the memory at the VM level**

Try increasing value to 2g, 3g, 4g, etc.

After 2g, you will probably have to increase memory on the VM.  Stop the VM and restart it with a different size with more memory.

## ****Starting and stopping the Docker cluster****

We will be covering Docker and Docker Clusters in weeks 4 and 5, with details about what every command means, variations of commands, etc.

For now, we will just be giving you a few simple commands to start and stop our first cluster.

Our first cluster will have two Docker Containers: one for Anaconda and one for Postgres. We will use Anaconda to run Jupyter Notebooks against a Python 3.x kernel, which will connect to our Postgres container.

Open a linux command line to the virtual machine.

Each cluster has its own directory. Change to the directory for this cluster:

cd ~/docker/clusters/anaconda\_postgres

If you have already run through this before, or have paused, verify that the cluster is not already running:

docker ps -a

If your cluster is not running, start it with the following command:

docker-compose up -d

Verify it's running with the following command:

docker ps -a

As long as your VM is running, you can safely keep the Docker cluster running. However, before stopping the VM, always shutdown all Docker clusters:

docker-compose down

Verify it's down with:

docker ps -a

## ****Find the external IP address of your virtual machine in Amazon Web Services:****

**AWS => N. Virginia => Services => Compute => EC2 => Instances => Instances => select instance => Connect => EC2 Instance Connect => Public IP address**

If you are using Windows, copy it to notepad.

If you are using Mac, copy it to TextMate. It is available to Berkeley students for free: <https://software.berkeley.edu/textmate>

## ****Start a Jupyter Notebook server in the Anaconda container:****

**Note: in the following command, simply copy and paste it "as is".  Do NOT replace with your IP address in this step:**

docker-compose exec anaconda jupyter notebook --notebook-dir=/user --ip='\*' --port=8888 --no-browser --allow-root

After you run the above command, you will get a URL to run Jupyter Notebooks from your web Browser.

Pull out the URL, copy and paste it to notepad (Windows) or TextMate (Mac).

**Change the IP address.**

Open an incognito Chrome browser and paste it in.

Be sure you are running Chrome!  We will be using software that uses Chrome extensions!

**To see what jupyter notebook severs are running and if other port 8888, kill them:**

docker ps

docker exec -it 791a19af5582 /bin/bash

jupyter notebook list

kill -9 604

**How to kill docker containers or clusters running:**

docker ps

docker stop $(docker ps -q)

# Relationship between Docker images, containers, and clusters

Understanding the hierarchical and sequential relationship between Docker images, containers, and clusters can indeed seem daunting at first. However, breaking down these concepts and seeing how they build upon each other can make the learning process more manageable. Here's a structured approach to help you grasp the whole ecosystem:

**1. Start with Docker Images**

* **Concept**: Think of Docker images as the foundational blueprints or templates for Docker containers. They are static files that package up an application and all its dependencies. Understanding Docker images is crucial because they are the starting point for running containers.
* **Analogy**: Consider a Docker image like a recipe for a dish. The recipe includes a list of ingredients (dependencies) and instructions (application code) on how to make the dish (run the application).

**2. Move to Docker Containers**

* **Concept**: A Docker container is a runtime instance of a Docker image. When you run an image, Docker creates a container from it. This container is a lightweight, standalone, executable package that runs the application in an isolated environment.
* **Analogy**: If a Docker image is a recipe, then running that recipe to make a dish is like starting a container. Each time you cook the dish using the same recipe, you create a new instance of the dish, just as each container is a new instance of an image.

**3. Understand Single Containers vs. Containers Outside Clusters**

* **Concept**: Running a single container is straightforward and suitable for simple applications or microservices. When you start having multiple containers that need to communicate or be managed together, you might still run them outside of a cluster for simplicity or development purposes. However, this lacks the orchestration and management features that come with clustering.
* **Analogy**: Cooking one dish (single container) for dinner is manageable. Cooking multiple dishes (containers outside clusters) for a large family dinner requires more coordination, but you might still do it on your own without formal tools or help.

**4. Grasp Docker Clusters and Orchestration**

* **Concept**: For applications that require scalability, fault tolerance, and easier management of many containers, Docker clusters come into play. Tools like Docker Swarm and Kubernetes help you manage these containers as a single entity, making it easier to deploy, scale, and maintain your application.
* **Analogy**: Think of a Docker cluster as hiring a team of chefs (the cluster) to prepare a large banquet (your application). Each chef (container) is responsible for their dish (service), but the head chef (orchestration tool) coordinates the overall meal preparation, ensuring dishes are prepared efficiently, on time, and according to guest needs (application requirements).

**Additional Contexts**

* **Development to Production**: Start by using Docker containers in development to encapsulate your application's environment. As your application grows or needs to be deployed, consider using Docker clusters for better resource management and scaling.
* **Networking and Storage**: Docker provides networking capabilities to connect containers to each other and to the outside world. Docker volumes and bind mounts help with data persistence and sharing between the host and containers.
* **Microservices Architecture**: Docker is particularly well-suited for microservices architectures, where each service runs in its own container, allowing for independent development, deployment, and scaling.

By understanding each component and how they build upon each other, you can better appreciate the Docker ecosystem as a whole. Starting from the basics (images and containers) and gradually moving to more complex concepts (clusters and orchestration) allows you to piece together the entire Docker puzzle. Remember, hands-on practice is key to solidifying your understanding, so consider setting up your Docker environments, running containers, and experimenting with Docker Swarm or Kubernetes to get a real feel for these technologies.

## Docker

Docker is a platform that allows users to easily develop, deploy, and run applications by using containers. Containers let you package your application with all of its dependencies into a standardized unit for software development, ensuring that your application will run the same regardless of the environment it runs in.

Now, let's go through each of the technologies you mentioned:

1. **Anaconda**: Anaconda is a distribution of Python and R programming languages for scientific computing, that aims to simplify package management and deployment. The official Anaconda Docker image would contain the Anaconda distribution pre-installed, making it easy to set up a scientific computing environment. The source code in this context would be the Dockerfile and any additional scripts used to build the Docker image.
2. **Postgres**: Postgres, or PostgreSQL, is a powerful, open-source object-relational database system. The official Postgres Docker image provides a pre-configured Postgres database server. The Dockerfile for Postgres would include instructions for installing Postgres and setting up the default configurations.
3. **Neo4j**: Neo4j is a highly scalable, native graph database purpose-built to leverage data relationships. The official Neo4j Docker image would include the Neo4j database and necessary configurations. The Dockerfile would contain instructions for installing Neo4j and setting it up.
4. **Mongo**: MongoDB is a document database with the scalability and flexibility that you want with the querying and indexing that you need. The official Mongo Docker image provides a MongoDB server. The Dockerfile for Mongo would detail the setup and configuration of a MongoDB instance.
5. **Redis**: Redis is an in-memory data structure store, used as a database, cache, and message broker. The official Redis Docker image includes the Redis server. The Dockerfile for Redis would contain the steps to install and configure Redis.
6. **Kafka**: Apache Kafka is an open-source stream-processing software platform. The official Kafka Docker image would contain Kafka and its dependencies, set up to start a Kafka server. The Dockerfile would specify how to install Kafka and configure it for initial use.
7. **Zookeeper**: Apache Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. The official Zookeeper Docker image includes the Zookeeper service. The Dockerfile would explain the setup and configuration for Zookeeper.
8. **NGINX**: NGINX is a web server that can also be used as a reverse proxy, load balancer, mail proxy, and HTTP cache. The official NGINX Docker image includes the NGINX software pre-configured. The Dockerfile would detail how NGINX is installed and set up within the container.

For each of these, the "**image**" refers to a Docker image, a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files.

The "**source code**" in this context usually refers to the Dockerfile and any associated scripts or files used to build that Docker image.

**Container**:

* + A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.
  + A container is essentially a lightweight, standalone, executable package of software that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and configuration files.
  + Containers are based on containerization, a technology that isolates applications from the rest of the system, ensuring that they work uniformly despite differences for instance between development and staging environments.

**Docker**:

* + Docker is a set of platform-as-a-service (PaaS) products that use OS-level virtualization to deliver software in packages called containers.
  + Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. It allows developers to package an application with all of its dependencies into a standardized unit for software development.
  + Docker is one of the most popular containerization platforms and has become synonymous with container technology. However, it's important to note that Docker is not the only container platform available.

In summary, a container is a lightweight, stand-alone, executable package of software that includes everything needed to run it, whereas Docker is a tool that facilitates the use of containerization technology by enabling the creation, deployment, and running of containers. Docker simplifies and accelerates your workflow, while containers ensure your applications run quickly and reliably from one computing environment to another.

## Lab: Docker - Creating Containers

This week focuses on Docker containers outside of clusters. Next week, we will focus on Docker clusters. Once you see Docker clusters, you will want to use them instead of single containers, however, we need to learn single containers first so we can understand clusters.

List containers (**does NOT list stray containers**):

docker ps 🡨 **Basically, it's a container that has the equivalent of, say, a blue screen. It's a container that's stopped or exited because it hit some error, and it's just sitting out there, but it's holding resources-- it's holding memory, it's holding resources, ports, TCP ports, things like that.**

**\*\* In the context of Docker and containerization, a "stray container" typically refers to a Docker container that is running but is no longer needed or has been forgotten.**

List containers (includes stray containers):

docker ps -a

List Docker networks:

docker network ls

These 3 networks are always there by default:

NETWORK ID NAME DRIVER SCOPE

22eac0bb47bb bridge bridge local

834615db7a14 host host local

3d4dfa7cfb57 none null local

**\*\* Bridge is your most common type of network. So we will be creating other networks with the bridge driver. So they just happen to name it bridge. Well, we're going to typically name them with the name of our cluster directory. Docker-compose will do that for us automatically. And then we're going to make bridge drivers, which are the most common drivers. There are several different types of drivers for networking.**

Create a container from the official Anaconda image.

* -it => i means to redirect standard input from the VM's bash shell to the container's bash shell; t means create a pseudoterminal to replace standard output
* --rm => means to remove the container when we exit the bash shell (prevents stray containers)
* continuumio is the Docker Hub repo name; just like GitHub has repos, Docker Hub has repos
* anaconda3 is the image name
* latest is the version; omitting a version defaults to latest

docker run -it --rm continuumio/anaconda3:2021.11 bash

**\*\* So remember, with Linux, a single dash means each letter is taken individually. So -it is the same as -i -t. Minus minus means the whole command is taken as a command line argument. So that --rm means it's not a -r -m, it's rm together.**

**\*\* So the image name has three pieces. It has Docker Hub repo. So just like GitHub has repos, Docker Hub has repos as well. So continuumio is the name of the company that makes Anaconda. So continuumio is the name of their Docker Hub. Then we have a slash, and then we have the image name, anaconda3-- so Anaconda for Python 3-- makes sense. And then a colon, and then the version. And if we omit the version, it assumes latest, but we went ahead and wrote it here. And if we assume-- if it's a standard image, we don't have to put repo name slash. For some of them that are standard images-- you'll notice Postgres, I believe, was that way-- we just said Postgres. We didn't have to say the Docker Hub, and we didn't have to give a version. If we don't give a version, they'll always give us the latest.**

**\*\* And how do I know I'm in a container? First thing you'll notice is the prompt looks different. So the prompt over here is telling me root. So I'm logged in as root to the Docker container. And we'll talk a little bit in class about the security issues around that, because Docker runs as root, and most containers we run as root. There are some containers we'll see that run as different users, which they solve the security problem, but they also create some new problems. So we'll talk about some of those issues in class. And then we have add and then the hostname. Well, your host name here is going to be basically the machine name.**

**A screenshot of a computer

Description automatically generated**

**Lab: Docker - Stray Container Cleanup**

Normally we won't get stray containers unless something goes wrong, such as shutting down the VM without shutting down Docker containers first, or an abend on Linux in the container. In this example, we will create stray containers on purpose, so we can see how to detect and clean them up.

See if we have any stray containers (remember -a includes stray containers):

docker ps -a

Create a container without the --rm option. When we exit the bash shell, we will be left with a stray container:

docker run -it continuumio/anaconda3:2021.11 bash 🡨 **never used this in real life**

**\*\* How I know it stray is because it'll say exited 13 seconds ago. If we had ports, which we'll see in a few minutes, it's hanging on to those ports even though it's exited. So exiting doesn't clean up everything. And it could also have memory or other resources that it's hanging on to.**

A computer code on a black background

Description automatically generated

To remove a container, we can remove them using either the container id or the container name:

docker rm [container\_id]

**or**

docker rm [container\_name]

If a container will not remove, such as a running container that won't shutdown, we can force it with the -f option. Forcing a remove should always be a last resort, as it can corrupt mounted file systems:

docker rm -f [container\_id]

or

docker rm -f [container\_name]

**\*\* if I've got a cluster with four or five containers running or even two containers running, one container crashes and the other doesn't, then I may have to forcibly kill the one that's running. Or sometimes just a container, we can't communicate with it, something went wrong, a piece of software in the container went wrong, and so we have to forcibly remove it.**

**Lab: Docker - Persistence, Directory Mounting to Containers**

Suppose we have a physical computer. We create files on that computer. Suppose we completely destroy the computer. We lose those files.

We create containers. We destroy containers. It's the equivalent of destroying a computer.

Suppose we plug an external hard drive into a physical computer. We can read files from the external hard drive. We can write files to the external hard drive. Suppose we destroy the computer, but keep the external hard drive. We can plug the external hard drive into another computer and read the files, so we don't lose them.

We can create containers with a mounted external directory. The mounted external directory works similar to an external hard drive. We can read files from the mounted external directory. We can write files to the mounted external directory. When we destroy the container, the files on the external directory will survive and can be accessed from the VM, and can be mounted to other containers we create.

docker run -it --rm continuumio/anaconda3:2021.11 bash 🡨 Any file that I create in the container is destroyed when the container is destroyed.

docker run -it --rm -v /home/w205/user:/user continuumio/anaconda3:2021.11 bash 🡨 **We're going to mount /home/w205/user to /user, OK? So the -v is volume mount or directory mount, I guess -v for Volume. So we're going to mount /home/w205/user to /user. So now when I go in, I'm going to have a persistent place, right?**

**OK, so if I do a pwd, I'm in /, but I can cd to /user, right? I can do a df -h, and I can see my mount. So do you see? I've mounted it to /user there. If a do a cd /user-- if I do an ls -l, look-- my how\_to\_guides, my labs, and my projects. Let's come over here. If I do a cd to home, ct to user, how\_to\_guides, labs, and projects. How\_to\_guides, labs, and projects-- see that? So this over here on the right is in the virtual machine because I'm on w205 at this IP address. And this over here is in the container, but I've mounted this /user-- /home/user if I do pwd here-- I've mounted /home/w205/user to /user over here, right? And as you can see-- why did I mount it that way? Why didn't I mount at /home/user? Because of directory permission issues. That's why we do it this way, because w205 owns this file and everything under it. It doesn't own the files above it, so we shouldn't mount anything above it. Sometimes, I get students who ask why we do that. So now, if I want, I could do something like that. I could just do an echo "Hello, World!" into my\_file. And if I cat my\_file-- OK, so my\_file.txt exists over here. And if I come over here to the virtual machine, oh, look, my\_file.txt is over here too. My\_file.txt, "Hello, World!" OK, so now let's go ahead and exit the container, which destroys it. We can verify it's destroyed by doing docker ps -a-- totally destroyed, container is gone. But my final lives on, right? So let me try to do this this way. So why don't we look over-- I can still do an ls -l. And there's my\_file.txt, and I'll do a cat on my\_file.txt.**

**When we do the volume mount, I was able to get any files that are in this directory or any directory underneath it. I can have access to those in the container. And then any files that I created can also come back,**

**Lab: Docker - Exposing and Mapping Ports in Containers**

In most cases we run TCP/IP, the protocal that runs the Internet. We are already familiar with IP addresses from AWS. IP addresses route us to a physical computer or VM. Once we get to the VM, a TCP port number is used to map us to a specific process. Think of a "TCP port" as a "TCP address". It's not a physical port, just a name for an address.

When we run Docker, things on the VM get a bit more complicated. We have to tell Docker which TCP ports to allow connections from the VM. Since we can have TCP port conflicts, it allows us to map a port to another port number.

Expose will allow another Docker container to connect using the port number. -p (ports) will allow the VM, and by extension external to the VM, to connect. -p also allow us to remap a port number if we have conflicts. Note the first port is external, followed by a colon, then the remap port:

docker run -it --rm --expose 8888 -p 8888:8888 continuumio/anaconda3:2021.11 bash

docker run -it --rm --expose 8888 -p 9999:8888 continuumio/anaconda3:2021.11 bash

**\*\* The minus p tells us it'll allow the VM, and by extension external to the VM, to connect. So if we want the virtual machine to be able to connect to our Docker container, we need to do the minus p here. The minus p also allows us to remap a port number if we have conflicts. So even if I do the expose here, and another container wants to connect, the weird thing is that the first port is the external, colon, and internal. And that's just the way networking works. Anything you do in networking, external comes first, and internal comes second. That's just the way it is. And we call this digest format, where we put colons between them. So always external number, colon, internal number. Everything I've seen related to networking, every protocol, that's just the way I guess double E's decided to do it for networking. And that's just the way it is.**



**so this basically is a wild card for IP version 4. And this is a wild card for IP version 6. The first two colons of the wild card for IP version 6, and then there's the colon and the mapping. So it's telling me that 888 for both IP version 4 and IP version 6, we're mapping port 888 out. I'm using the protocol, this tells me it's using TCP.**

**Lab: Docker - Stray Network Cleanup**

We have seen that Docker provides 3 networks by default:

docker network ls

NETWORK ID NAME DRIVER SCOPE

22eac0bb47bb bridge bridge local

834615db7a14 host host local

3d4dfa7cfb57 none null local

When we create a Docker container, it will use the bridge network by default. We can create and use our own network, typically of type bridge (network types is a very complicated subject we won't cover here).

docker network create -d bridge my\_network

**\*\* d is the type or the driver, and then my network.**

Start a container and tell it to use this network:

docker run -it --rm --network=my\_network continuumio/anaconda3:2021.11 bash

Prune networks that are not in use:

docker network prune

If prune does not work, we can force remove the network (only as a last resort!):

docker network rm my\_network

**Lab: Docker - Microservices**

In the concepts, we covered how originally services were huge due to technology limitations, but now with more modern tecnology, it's technologically feasible to have smaller and smaller services, to the point of microservices.

We also covered how corruption is worse than a crash. We covered the example of a cell phone starting to exhibit quirky behavior, we reboot it, and it works fine again.

We also covered business cases were we created a container for each individual user upon login, kept each user in their own container for the duration of the login, and destroyed the container upon logout.

**Up until now, we have been running the bash shell when we create a container. We can actually run any program.**

The following example: creates a Docker container, run the find program, exits, and destroys the container:

docker run -it --rm continuumio/anaconda3:2021.11 find /var/log

**\*\* We've been putting bash here, but we could run any program we want. So here's an example here. We're going to create a Docker container.**

**\*\* here find /var/log is the program (just a finding /var/log)**

**Lab: Docker - Creating an Image**

List all images:

docker images

**\*\* we're going to look at Docker, creating an image. So we've been using these canned images that have been coming off of Docker hub, or they're either official images. So now for this class, we've created quite a few custom images. If you'll notice, all of our images start w205\_, and I've got one, two, three, four, five, six, seven, eight different images here, Anaconda, Kafka, Manga, Mongo, Neo4j, NGINX, Postgres, Redis, and Zookeeper. So all of these have, I'm taking an official image, and I'm doing some customization to it. And the way we do that is we create a little Docker file. So each one has a directory with at least a Docker file, plus any other files that we need to copy in there. And I'll actually run through the build.**

**\*\* you notice red is postgres, NGINX, Mongo, and Neo4j These are all official images that don't come from a Docker hub. Continuum I/O puts Anaconda in their own Docker hub, because it's got Docker hub slash. And then Confluent makes Kafka and Zookeeper, or they make Kafka, and they also have a version of Zookeeper. And so Confluent Inc has their own Docker hub in this, but all these others are official images, because you don't see the Docker hub name on them.**

Each Docker image we are using for this course is custom build from a standard Docker image. Each image has a directory. The directory hold a Dockerfile, plus any files that we need to copy into the image:

cd ~/docker/images/w205\_anaconda

cd ~/docker/images/w205\_kafka

cd ~/docker/images/w205\_mongo

cd ~/docker/images/w205\_neo4j

cd ~/docker/images/w205\_nginx

cd ~/docker/images/w205\_postgres

cd ~/docker/images/w205\_redis

cd ~/docker/images/w205\_zookeeper

**Warning - you may just want to watch the video for creating images if this is new for you, as a lot can go wrong with image building. Or, as an alternative, create a second VM, try this in the second VM, and then delete the second VM.**

To remove an image:

docker image rm xxxxx

Remove all images (be careful):

docker image rm $(docker images -aq)

To rebuild the custom images for this course:

cd ~/docker/images/w205\_anaconda

docker build -t w205\_anaconda .

cd ~/docker/images/w205\_kafka

docker build -t w205\_kafka .

cd ~/docker/images/w205\_mongo

docker build -t w205\_mongo .

cd ~/docker/images/w205\_neo4j

docker build -t w205\_neo4j .

cd ~/docker/images/w205\_nginx

docker build -t w205\_nginx .

cd ~/docker/images/w205\_postgres

docker build -t w205\_postgres .

cd ~/docker/images/w205\_redis

docker build -t w205\_redis .

cd ~/docker/images/w205\_zookeeper

docker build -t w205\_zookeeper .

**\*\* And then we're going to do a Docker build. So we say Docker build minus t with the image name, and then dot means current directory. Be sure you get the dot. Hopefully, you're probably not doing this unless you're doing it into a, unless you're doing it into a second virtual machine that you're going to throw away.**

**\*\* "docker build -t w205\_anaconda ."**

**docker build: This is the main command used in Docker to build a new image from the instructions in a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.**

**-t: This flag stands for "tag". It allows the user to assign a name (and optionally a tag in the 'name:tag' format) to the image that is being built. This is useful for identification and reference purposes.**

**w205\_anaconda: This is the name (and optionally tag) that you are assigning to your new Docker image. In this case, "w205\_anaconda" could be a name chosen to represent an image that perhaps has the Anaconda distribution installed, relevant to a course or project labeled 'W205'.**

**.: This indicates the build context to the Docker daemon. In this case, the . denotes the current directory. The build context typically contains a Dockerfile and any necessary files to build that Docker image. When you specify ., it tells Docker to use the Dockerfile in the current directory and to use the current directory as the context for any instructions within the Dockerfile that require file system context (like COPY or ADD commands).**

**So, putting it all together, the command "docker build -t w205\_anaconda ." is instructing Docker to build a new image using the Dockerfile in the current directory, and to tag the resulting image with the name "w205\_anaconda". This image can then be used to create containers that will run whatever application or environment is specified in the Dockerfile**

**Lab: Docker - How Various Official Images are Built**

**Docker Hub** - stores the Docker images in the format: **docker\_hub\_repo\_name/image\_name/version\_name**

**\*\* so there's a Docker Hub that the images are stored in, and then there is how the images are created below. So the source code for every create of every official image, or every image in Docker Hub, is also available. And those are available, and they're typically stored in the GitHub repo. I say typically. Actually, I don't think I've ever seen one that wasn't in the GitHub repo. Of course, if you make your own Docker Hub repo, there's no obligation for you to publish the source code. But for all these official images, they always publish them.**

To create these Docker images, the source code, including Dockerfile and any files that will be copied in are typically stored in a GitHub repo. It should always be possible to clone down the GitHub repo and build any image from scratch.

Here are some example official images we can take a look at:

**Anaconda**

[https://hub.docker.com/r/continuumio/anaconda3Links to an external site.](https://hub.docker.com/r/continuumio/anaconda3)

[https://github.com/ContinuumIO/docker-imagesLinks to an external site.](https://github.com/ContinuumIO/docker-images)

**Postgres**

[https://hub.docker.com/\_/postgresLinks to an external site.](https://hub.docker.com/_/postgres)

[https://github.com/docker-library/postgresLinks to an external site.](https://github.com/docker-library/postgres)

**Neo4j**

[https://hub.docker.com/\_/neo4jLinks to an external site.](https://hub.docker.com/_/neo4j)

[https://github.com/neo4j/docker-neo4jLinks to an external site.](https://github.com/neo4j/docker-neo4j)

**Mongo**

[https://hub.docker.com/\_/mongoLinks to an external site.](https://hub.docker.com/_/mongo)

[https://github.com/docker-library/mongoLinks to an external site.](https://github.com/docker-library/mongo)

**Redis**

[https://hub.docker.com/\_/redisLinks to an external site.](https://hub.docker.com/_/redis)

[https://github.com/docker-library/redisLinks to an external site.](https://github.com/docker-library/redis)

**Kafka**

[https://hub.docker.com/r/confluentinc/cp-kafka/Links to an external site.](https://hub.docker.com/r/confluentinc/cp-kafka/)

[https://github.com/confluentinc/kafka-imagesLinks to an external site.](https://github.com/confluentinc/kafka-images)

**Zookeeper**

[https://hub.docker.com/r/confluentinc/cp-zookeeperLinks to an external site.](https://hub.docker.com/r/confluentinc/cp-zookeeper)

Same GitHub repo as Kafka above

**NGINX**

[https://hub.docker.com/\_/nginxLinks to an external site.](https://hub.docker.com/_/nginx)

[https://github.com/nginxinc/docker-nginxLinks to an external site.](https://github.com/nginxinc/docker-nginx)

Lab: Docker Compose - Single Container Cluster

Each Docker cluster has its own directory. In each directory is a file docker-compose.yml. Here is an example of single container cluster:

cd ~/docker/clusters/anaconda

Review docker-compose.yml

A screen shot of a computer

Description automatically generated

\*\* And each directory is a file called docker-compose.yml. And we pronounce that "yamel." You could also put why YAML. So you could change the name of that. I really wouldn't recommend it. Your coworkers may hate you forever. If you're at a company and you change the name of the Docker Compose file, you can change it and tell Docker to use a different file than docker-compose.yml, but I really wouldn't recommend that. I have students who want to do that sometimes, but I'm thinking the reason for that is that if somebody is trying to look at your work, you leave the company, or just somebody else is looking at your work, then you're probably going to get phone calls saying, I can't find the docker-compose file. Where'd you put it? So I would just put the Docker file, I would keep with their standard naming. I've never seen anything work anywhere where somebody's named it something different. So I would stick with the basics here on this. That way people can find your work and not call you and bug you. So let's come over here to clusters. Actually, instead of, you could just cd into it, but let me go piecemeal. So let's go with Docker. So remember with Docker, we have three directories. We have one for clusters that we're going to use, and then remember, we've already seen images. And then mounts are going to be where our mounts are located, and we'll look at those as we go through these clusters. So if we go into clusters, and then here's a list of all of our clusters here. So we'll look at all of these we've got. Obviously, what I did is I just put the names of all the products that they're using. So our smallest one would be just Anaconda by itself, and then our largest one would be Anaconda, Postgres, Redis, and NGINX. So there you would have a four-container cluster. So you can count these, and so all of them have Anaconda, because we pretty much are using Python and Jupyter Notebooks, and that's where we're running them. We're running our Jupyter Notebook server in Anaconda. And the one that you've been using so far, starting in Week Two, is this Anaconda Postgres. So that's the one you've been familiar with. It has an Anaconda container in it and has a Postgres container in it. OK, so we'll go ahead, and I'll go ahead and just cd, use this command here. There we are. OK, now you notice I have a single file in here called docker-compose.yml. So I could cat out docker-compose.yml. And we can take a quick look at it. So first thing is telling the version that it is, and then you'll notice that we have things like services here. And so we talked about microservices. It used to say containers, and they're trying to say services now, because they're really trying to play on the micro container, using containers as microservices. So they've started doing that. So this is the name of the container, will be Anaconda. So I'm going to call it Anaconda. And the image is the one that we created, and we went through that last week, how I created the custom image. So I have a custom image here, w205\_anaconda. This is the same as the minus i and the minus t on the command line. And then volume mounts, so this is the same as the minus v on the command line. And then the first thing that we'll do is we'll mount a /home/w205/user. And we did that last week in the single container. The reason there is that everything that's under my home w205 user directory, all of the stuff, your projects, everything can be mounted, and we'll have access to it. And then we also have, Anaconda may create environments. So we're going to put this in Docker Mounts Anaconda. And let's look over at that directory over here. This one's going to be a little bit trickier. So if I'm at home here, so I'm in my tilde directory, and I cd into Docker, and then I cd into mounts, then you'll notice that all of the mounts are, I've put them based on the product that they are. So Anaconda mounts are going to be in Anaconda. So you notice that it has a directory called environments. So what we're seeing here is that this is that directory right here. So if I do a PWD here-- let's get this one over a little bit-- you notice that mount, this right here matches. Actually, the directory environment matches. If we cd into directory environment, actually, I don't know if-- so there's nothing in there. I haven't actually created any environments. But if we did, if you wanted to create your own local environments, this way they'll outlive. So the purpose of the mounts are, so remember, we talked about last week, so that we can pass files in and out. But in this case here, it's going to store any environments that you create. It's going to write them to opt/conda/envs, and then that's going to get written to this directory right here. Notice that we have ownership issues here, because it creates the environment directory as root. That's because Docker runs as root, so it created those. So I'll need to use sudo if I want to touch those, and we want to be sure that we don't change the owner nor the permissions on any of these files. So anything under mounts, and if you notice in my fixed permissions scripts, so basically all of these directories can be owned by w205 with the mids group. But anything underneath these, you do not want to change any permissions or directories, because it can force an image to quit working. So just be careful. These are OK, and these are fine the way they are. But anything below any of these, you don't want to change. So let's see, did we cover everything in our Docker? OK, then obviously, we have our expose for 8888, which everyone should know is Jupyter Notebook. And then we're mapping external 8888 to internal 8888. Remember, external always comes first in networking. So that's it there.

Check for stray containers:

docker ps -a

Startup our Docker cluster. Note that we must be in the directory that contains the docker-compose.yml file for any command that starts with docker-compose.

docker-compose up -d

Check that the cluster came up (Note that there is no -a option with docker-compose):

docker-compose ps

docker ps -a

To exec a bash shell into a container (Note that we can run other programs besides bash):

docker-compose exec anaconda bash

\*\* Anaconda and run this program here, which is a bash shell. So I'm saying exec into the Anaconda and run the bash shell. So let's try that.

Check the linux virtual console (aka "logs") for the container:

docker-compose logs anaconda

See the network:

docker network ls

\*\* the one that's named bridge using the bridge driver is the one that any container, any single container defaults to. And then if you remember last week, when we talked about single containers, we created our own network, and we wrote a container that would connect to that network. Well, one of the things that docker-compose does is it always creates a default network, and it's going to be the same name as the directory. So it's going to be directory\_default, and it's going to give the driver as bridge. Because it's a bridge type driver, and we won't go into all the different types of drivers we can have, but bridge is your most common one and the one that we'll use for this course.

Shutdown (aka "tear down") the cluster:

docker-compose down

\*\* And remember, if I say docker-compose, I have to be in the directory with the docker-compose.yml.

Verify the cluster is down cleanly with no stray containers nor stray networks:

docker-compose ps

docker ps -a

docker network ls

Lab: Docker Compose - Stray Cluster and Stray Network Cleanup

When running a cluster, there are several ways we can get a stray container or stray network:

* In a multi-container cluster, a container crashes and the other containers are still running
* We run multiple clusters and there is a resource conflict between them which prevents some containers from booting, while other containers boot
* We stop a container that is part of a cluster (accident?)
* We shutdown a cluster with the wrong docker-compose.yml file which is unable to stop all containers
* We stop the VM while a cluster is running

To detect stray containers in a cluster:

docker-compose ps

docker ps -a

If we see that a container in a cluster is not working, first try the cluster shutdown, as that would be the cleanest solution:

docker-compose down

Recheck for stray containers. If any are found, we should remove them like this:

docker rm container\_name

If the container does not respond, try a forced removal:

docker rm -f container\_name

Now check for stray networks:

docker network ls

Three default networks are always present:

NETWORK ID NAME DRIVER SCOPE

22eac0bb47bb bridge bridge local

834615db7a14 host host local

3d4dfa7cfb57 none null local

Try to first prune any stray networks:

docker network prune

\*\* any networks that are not in use, it'll will clean them up. So you always want to do containers first, and then prune any stray networks.

If that does not work, then as a last resort, remove the network:

docker network rm network\_name

Lab: Docker Compose - Two Container Clusters

Here are the two container clusters:

cd ~/docker/clusters/anaconda\_mongo

cd ~/docker/clusters/anaconda\_neo4j

cd ~/docker/clusters/anaconda\_postgres

cd ~/docker/clusters/anaconda\_redis

Use the same commands we previously learn to start the cluster, verify it's running, verify the network, exec a bash shell into the containers, shutdown the cluster, verify it came down cleanly:

docker ps -a

docker network ls

docker-compose up -d

docker-compose ps

docker-compose exec xxxx bash

docker-compose logs xxxx

docker-compose down

Lab: Docker Compose - Networking in a Cluster Environment

Review the networking parameters in the various docker-compose.yml files:

cd ~/docker/clusters/anaconda

cd ~/docker/clusters/anaconda\_kafka\_zookeeper

cd ~/docker/clusters/anaconda\_mongo

cd ~/docker/clusters/anaconda\_neo4j

cd ~/docker/clusters/anaconda\_postgres

cd ~/docker/clusters/anaconda\_postgres\_neo4j

cd ~/docker/clusters/anaconda\_postgres\_redis

cd ~/docker/clusters/anaconda\_postgres\_redis\_nginx

cd ~/docker/clusters/anaconda\_redis

Lab: Docker Compose - Multiple Container Clusters with Networking

Here are the multiple container clusters (more than 2 containers):

cd ~/docker/clusters/anaconda\_kafka\_zookeeper

cd ~/docker/clusters/anaconda\_postgres\_neo4j

cd ~/docker/clusters/anaconda\_postgres\_redis

cd ~/docker/clusters/anaconda\_postgres\_redis\_nginx

Use the same commands we previously learn to start the cluster, verify it's running, verify the network, exec a bash shell into the containers, shutdown the cluster, verify it came down cleanly:

docker ps -a

docker network ls

docker-compose up -d

docker-compose ps

docker-compose exec xxxx bash

docker-compose logs xxxx

docker-compose down

**Github and git commands**

**\*\* branch-name = ‘project’ in this case:**

**# Note that the token (password) will not echo (print to the screen) when you copy it.**

git clone xxxxx

**# List the contents of the directory to see the directory name containing the clone:**

ls -l

**# Change to the repo directory. Obviously, replace the yyyy's with the name of your repo. In GitHub classroom, it will be different for each student, so we cannot give you a specific name here:**

cd yyyy

**# Show the status of the repo. You will use this command numerous times:**

git status

**# Create a branch called project. Please use this exact name so the graders can easily find your branch!**

git branch project

**# Checkout the project branch (Switch to your target branch)**

git checkout branch-name

**# Update your branch**

git pull origin branch-name

**# See what you have changed:**

git status

**# Stage your changes for commit**

git add filename

**# Or, to add all changed files**

git add .

**# Commit your changes with a message**

git commit -m "A descriptive message about your update"

**# Verify the commit:**

git status

**# Push your changes to GitHub**

git push origin branch-name