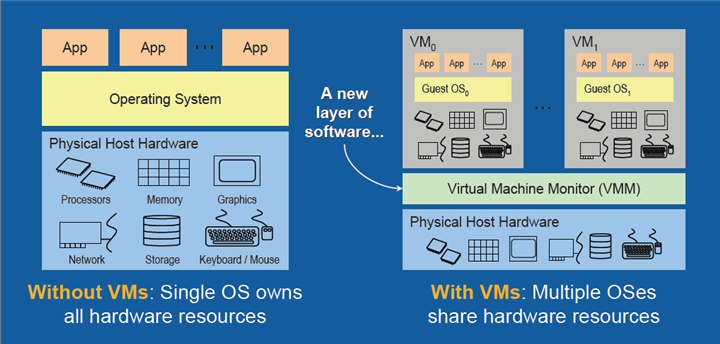
**Agenda: Introduction to Containers and Docker**

* Understanding VM's and Containers.
* What is Docker?
* Docker Benefits.
* Docker Architecture and Docker Taxonomy.

**Understanding Virtual Machines and Containers**

**What is a Virtual Machine**

* A virtual machine is a computer file, typically called an image, which behaves like an actual computer – computer within a computer
* It runs in a window, much like any other programme, giving the end user the same experience on a virtual machine as they would have on the host operating system itself.
* Multiple virtual machines can run simultaneously on the same physical computer.
* Hypervisor is the software required for managing VM, emulates the PC or server's CPU, memory, hard disk, network and other hardware resources completely, enabling virtual machines to share the resources.
* The hypervisor (Hyper-v, VMWare, Xen, KVM) can emulate multiple virtual hardware platforms that are isolated from each other, allowing virtual machines to run Linux and Windows Server operating systems on the same underlying physical host.
  + Type 1 Examples: Hyper-V, Xen, VMware ESXi
  + Type 2 Examples: Virtual Box,, VMWare Workstation, KVM



**What is a Container?**

* Containerization is an **approach** to software development in which **an application or service, its dependencies, and its configuration** are packaged together as a **container image**. You then can test the containerized application **as a unit** and deploy it as a container image instance to the host operating system.
* Placing software into containers makes it possible for developers and IT professionals to deploy those containers **across environments** with little or no modification.
* Containers also isolate applications from one another on a **shared operating system (OS**). Containerized applications run on top of a **container host**, which in turn runs on the OS (Linux or Windows). Thus, containers have a significantly smaller footprint than virtual machine (VM) images.
* Con, agility, scalability, and control across the entire application life cycle workflow. The most important benefit is the isolation provided between Dev and Ops.

**Containers vs VMs**

**Containers are lightweight** because they don’t need the extra load of a hypervisor, but run directly within the host machine’s kernel. This means you can run more containers on a given hardware combination than if you were using virtual machines. You can even run Docker containers within host machines that are actually virtual machines!

Graphical user interface

Description automatically generated with medium confidence

Container = VM - Guest OS

**What is Docker and its Benefits**

**What is Docker**

Docker is an open platform for developing, shipping, and running applications as containers.

**Benefits of Docker / Container:**

* **Resource Efficiency**: Docker is lightweight and fast. Process level isolation and usage of the container host’s kernel is more efficient when compared to virtualizing an entire hardware server using VM.
* **Fast and consistent delivery of your applications:** By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

**Consider the following example scenario:**

1. Your developers write code locally and share their work with their colleagues using Docker containers.
2. They use Docker to push their applications into a test environment and execute automated and manual tests.
3. When developers find bugs, they can fix them in the development environment and redeploy them to the test environment for testing and validation.
4. When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.

**Docker is available for implementation across a wide range of platforms:**

* **Docker Desktop**: Mac OS, Windows 10/11, Linux.
* **Docker Server**: Various Linux distributions and Windows Server 2019/2022.
* **Cloud**: Amazon Web Services, Google Compute Platform, Microsoft Azure, IBM Cloud, and more.

**Open Container Initiative (OCI)**

* The OCI is an open governance structure for the express purpose of creating open industry standards around container formats and runtimes
* Docker started it
* They have two specs: runtime-spec and image-spec
* This deals with containers in the abstract

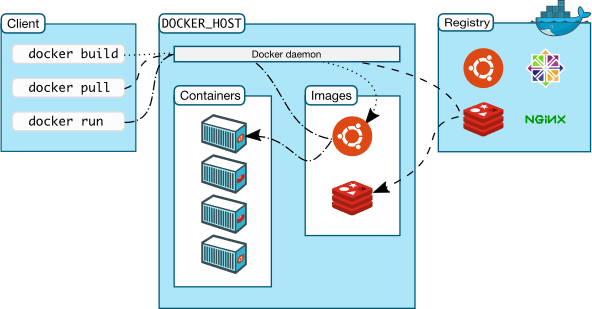
**Alternatives to Docker**

1. Podman by RedHat
2. Rkt from CoreOS
3. Mesos from Apache

**Docker Architecture and its Taxonomy**

***Docker Engine*** is a client-server application with these major components:

* A server which is a type of long-running program called a **daemon** process.
* A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
* A command line interface (CLI) client (the docker command).

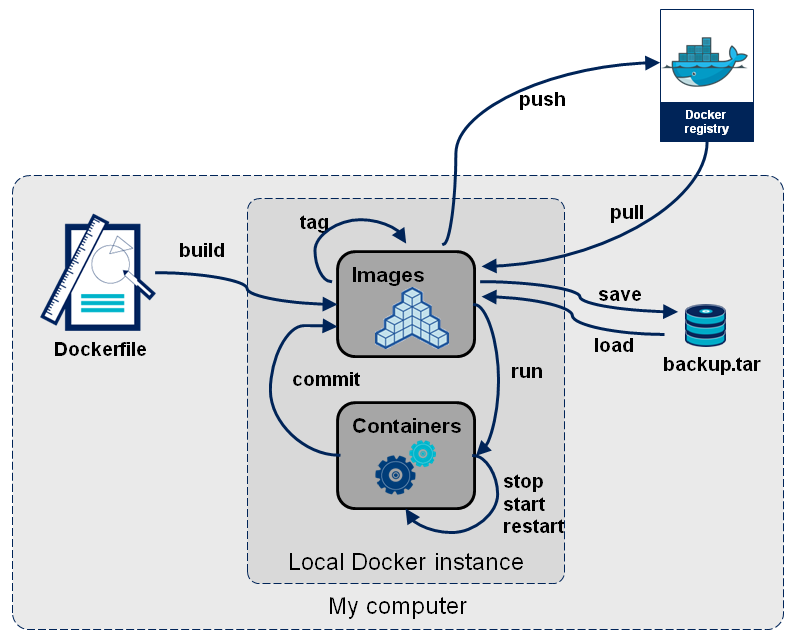
 ****

**Docker Daemon:**

* The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as **images, containers, networks, and volumes**.
* A daemon can also communicate with other daemons to manage Docker services.

**Docker client:**

* The Docker client (docker) is the primary way that many Docker users interact with Docker.
* When you use commands such as ***docker run***, the client sends these commands to Docker Daemon, which carries them out.
* 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 sockets or a network interface.
* The Docker client can communicate with more than one daemon.



**Docker Commands**

**Listing all images from local registry:**

* docker version
* docker pull hello-world
* docker pull nginx
* docker pull alpine
* docker pull ubuntu
* docker image history nginx
* docker inspect nginx
* docker images
* docker image ls
* docker images --no-trunc (List full lengh image IDs)
* docker images --filter=reference=alpine

**Some things to note:**

1. IMAGE ID is the first 12 characters of the true identifier for an image. You can create many tags of a given image, but their IDs will all be the same (as above).
2. VIRTUAL SIZE is virtual because it's adding up the sizes of all the distinct underlying layers. This means that the sum of all the values in that column is probably much larger than the disk space used by all of those images.
3. The value in the REPOSITORY column comes from the -t flag of the docker build command, or from docker tag-ing an existing image. You're free to tag images using a nomenclature that makes sense to you, but know that docker will use the tag as the registry location in a docker push or docker pull.
4. The full form of a tag is [REGISTRYHOST/][USERNAME/]**NAME**[:TAG]. For ubuntu above, REGISTRYHOST is inferred to be registry.hub.docker.com. So if you plan on storing your image called my-application in a registry at docker.example.com, you should tag that image docker.example.com/my-application.
5. The TAG column is just the [:TAG] part of the full tag. This is unfortunate terminology.
6. The latest tag is not magical, it's simply the default tag when you don't specify a tag.
7. You can have untagged images only identifiable by their IMAGE IDs. These will get the <none> TAG and REPOSITORY. It's easy to forget about them.

**Basic Docker Commands:**

1. docker pull hello-world
2. docker container run hello-world
3. docker ps
4. docker ps -a
5. docker run hello-world
6. docker run --name **hello** hello-world
7. docker rm <container-id>
8. docker run --rm hello-world
9. docker container prune
10. docker run alpine ls -l
11. docker run alpine echo "Hello from Sandeep"

**To create a container in Interactive Mode**

1. docker run **-it** alpine /bin/sh
   1. ls -l
   2. echo "hello" > hello.txt
   3. ls
   4. exit
2. docker ps -a
3. docker start <container ID>
4. docker attach <container ID>

Ctrl + P + Q to detach from the shell, switching to host.

1. docker **top** <container id> #Display the running processes of a container

**To run a sample web application:**

docker run -p 8001:80 nginx

# Start a new Terminal / Command Window

docker ps

docker inspect <container-id>

curl <container-ip> - Fails to connect because container is in network with WSL or Linux VM Host machine. And NOT with our local machine.

curl localhost:8001

docker run **-d** --rm -p 8080:80 nginx

docker logs <container-id>

* -p publishes a port. The IIS image is built to allow traffic in on port 80. This maps port 8080 on the host to port 80 in the container
* -d starts in detached mode, Docker runs the container in the background and monitors it

**Exec Command**

* Start a new container from the nginx image on Docker Hub
* Using exec, get into the container and have a look around
* Visit the web site and make sure you see the default home page.
* Now go change the home page index.html file located in **/usr/share/nginx/html**.
* Add file **/usr/share/nginx/html/hello.html (use echo)**
* Verify that your changes show in the web site.

**Solution:**

1. docker run -p 8080:80 -d --rm nginx
2. docker ps
3. **docker exec -it <container name> /bin/sh**

# cd /usr/share/nginx/html

# echo "hello" > "hello.html"

# exit

1. curl <http://localhost:8080>/hello.html
2. docker **commit** <container id> myngnix
3. **docker run -p 8081:80 mynginx**
4. **curl** [**http://localhost:8081/hello.html**](http://localhost:8081/hello.html)
5. **docker diff <container-id>**

**Creating Image using Dockerfile**

* A Dockerfile is a text file which contains a series of commands or instructions. You need to use an existing image as the starting point for your app, but you decide which one.
* These instructions are executed in the order in which they are written.
* Execution of these instructions takes place on a base image.
* On building the Dockerfile, the successive actions form a new image from the base parent image.

1. **Create a d:\Demo\Dockerfile:**

FROM nginx

WORKDIR /usr/share/nginx/html

RUN echo "Hello, How are you…" > hello.html

COPY . .

1. The first instruction must be **FROM**. This instruction initializes a new build stage and sets the Base Image for the remaining instructions. The multi-arch tags pull either Windows or Linux containers depending on the Docker for Windows container mode.
2. The **WORKDIR** instruction sets the working directory. If the directory doesn't exist, it's created. In this case, WORKDIR is set to the HelloWorld directory. The WORKDIR instruction **wont create a new layer** in the image but will add metadata to the image config.
3. The **COPY** instruction copies new files or directories from the source path and adds them to the destination container filesystem.
   1. COPY <src>... <dest>
   2. COPY ["<src>",... "<dest>"] (this form is required for paths containing whitespace)
4. **RUN** executes the instruction

**Build the Image**

docker **build** -t mynginx:v1 .

Note: Image names must be unique and are specified in the format <username>/<repository>:<tag>.

**Run the Image – Create the container**

docker **run** sandeep/mynginx:v1

**Show the running process in the container (Container must be running and not exited)**

D:\demo>docker top <Container ID>