Docker Essentials: Behind The Scene

As developer, while we try to run an existing code base, we often have to troubleshoot environment issues. This could be dependency issue, module installation problem or environment mis-match.

Docker, in its core is trying to fix these problems. Docker is trying to make it super easy and really straight-forward for anyone to run any code-base or software in any pc, desktop or even server.

In a nutshell, `Docker` make it really easy to install and run software without worrying about setup and dependencies.

> According to wikipedia, `Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers.` Just like we can manage our application, docker enables us to manage the application infrastructure as well.

How OS runs on machine?

Let's get familiar with couple of OS components,

Kernel : Most OS has a kernel. It runs the software processes that govern the access between all the programs running on the computer and all the physical hardware connected to the computer.

For example, If we write a file in physical hard-disk using node.js, it's not node.js that speaking directly to the physical device. Actually node.js make a system call with necessary information to the kernel and the kernel persist the file in the physical storage.

So the kernel is an intermediary layer that govern the access between the program and physical devices.

System Call : The way program pass instruction to the kernel is called system call. These system calls are very much like function invocation. The kernel exposes different endpoint as system call and program uses these system call to perform an operation.

For example, to write a file, the kernel expose an endpoint as system call. Program like node.js invoke that system call with necessary information as parameters. This parameters can contain the file name, file content etc.

A Chaos Scenario : Lets consider a situation, where there are two program that require two types of node.js runtime.

- Program 01 (Require Node.js 8.0)
- Program 02 (Require Node.js 12.0)

Since, we can not install these 2 versions in runtime, only one program can work at a time.

A Solution With Namespace : OS has a feature to use namespace. Using namespace, we can segment hard-disk in multiple parts. For example, to use two versions of Node.js during runtime, one of the segment will contain node.js runtime version of 8.0 and another segment will contain node.js runtime version 12.0

In this way, we can ensure the `program 01` will use the segment of node.js with 8.0 runtime and the `program 02` will use the segment of node.js 12.0 as runtime.

Here the kernel will determine the segment during system call by the programs. So for `program 01` the kernel drag the system call from segment with 8.0 node js and for `program 02` the kernel will drag the system call from the segment with 12.0 node.js runtime.

> Selecting segment„ using system call, based on the program is called namespacing. This allow isolating resources per processes or a group of processes.

Namespace can be expanded for both cases

- Hardware Elements
- Physical Storage Device
- Network Devices
- Software Elements
- Inter process communications
- Observer and use of processes

Control Group : Also known as `cgroup`. This limits the amount of resources that a particular process can use. For example it determine,

- Amount of memory can be used by a process
- Number of I/O can be used by a process
- How much CPU a process can use
- How much Network bandwidth can a process use

Namespacing Vs cgroup : `Namespacing` allow to restrict using a resource. While the `cgroup` restrict the amount of resource a process can use.

Utilizing Namespacing in The Docker Container

Docker container is a group of

- Process or Group of processes
- Kernel
- Isolated resource

In a `container` kernel observes the system call and for a process, guide to the specified `isolated resources`.

> The windows and Mac OS does not have the feature of `namespacing` and `cgroup`. When we install docker in these machines, a linux virtual machine is installed to handle the isolation of resources for the processes.

Relations Between Image and Container : An image have the following components

- File System snapshot
- Startup commands

When we take an image and turn it into a container, the kernel isolated the specified resource just for the container. Then the process or file system snapshot takes places in the physical storage.

While we run the startup commands, it installed these process from the physical storage and start making the `system call` using the `kernel`.

Docker Ecosystem

Dockers ecosystem contains

- Docker Client
- Docker Server
- Docker Machine
- Docker Image
- Docker Hub

- Docker Compose

Docker Image : Single file with all the dependencies and config required to run a program.

Docker Container : Instance of the `Docker Image`.

Docker Hub : Repository of free public `Docker Images`, can be downloaded to local machine to use.

Docker Client :

- Took the command from the user
- Do the pre-processing
- Pass it to the `Docker Server`
- `Docker Server` do the heavy processing

An Docker Example :

i don't want you to be disappointed, Assuming you have already installed docker in your system, let's run,

```
docker run hello-world
```

- It imply to run an `container` from the image `hello-world`
- This `hello-world` is a tiny little program, whose sole purpose is to print `Hello from Docker!`
- `Docker Server` check the `local image cache`. If it is not exist in the `local image cache` it goes to `Docker Hub` and download the `image`.
- Finally the `Docker Server` run the `image` as `container` or `image instance`.
- If we run the same command again and the `image` is already in the cache, It does not download it from the `Docker Hub`.

Docker Lifecycle

When we run a docker container using `run` command,

```
docker run image_name
```

Then, the `docker run` is equivalent to the following 2 commands:

1. `docker create`
2. `docker start`

With `docker create`, the `file system snapshot` of the image is being copied to `isolated physical storage`.

Then with `docker start` we start the `container`.

Example :

Let's do the hands on what we are claiming with a image `hello-world`.

```
docker create hello-world
```

This will return the `id` of the created container.

Using the `id` we can now start the docker.

```
docker start -a id
```

This will give us the output `Hello from Docker!`.

Here the `-a` flag watch out the `container` output and print it in `console`.