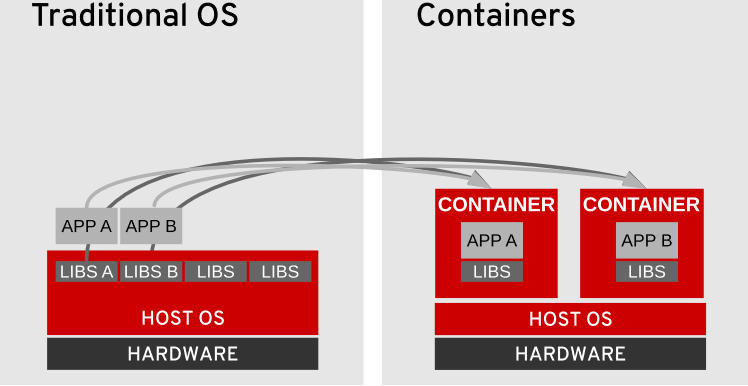
**What is container?**

A **container** is a runtime instance of an image–what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, docker ps, just as you would in Linux.

****

**Advantages of Containers**

1. **Low hardware footprint**
2. **Environment isolation**
3. **Quick deployment**
4. **Multiple environment deployment**
5. **Reusability**

**Docker Architecture**

Docker is one of the container implementations available for deployment and supported by companies such as Red Hat in their Red Hat Enterprise Linux Atomic Host platform. Docker Hub provides a large set of containers developed by the community.

Docker uses a client-server architecture, described below:

***Client***

The command-line tool (**docker**) is responsible for communicating with a server using a RESTful API to request operations.

***Server***

This service, which runs as a **daemon** on an operating system, does the heavy lifting of building, running, and downloading container images.

The daemon can run either on the same system as the **docker** client or remotely.

**Docker Core elements**

***Images***

Images are read-only templates that contain a runtime environment that includes application libraries and applications. Images are used to create containers. Images can be created, updated, or downloaded for immediate consumption.

***Registries***

Registries store images for public or private use. The well-known public registry is Docker Hub, and it stores multiple images developed by the community, but private registries can be created to support internal image development under a company's discretion. This course runs on a private registry in a virtual machine where all the required images are stored for faster consumption.

***Containers***

Containers are segregated user-space environments for running applications isolated from other applications sharing the same host OS.

Reference :[Docker Hub website](https://hub.docker.com/" \t "_blank)

## Containers and the Linux Kernel

Containers created by Docker, from Docker-formatted container images, are isolated from each other by several standard features of the Linux kernel. These include:

***Namespaces***

The kernel can place specific system resources that are normally visible to all processes into a namespace. Inside a namespace, only processes that are members of that namespace can see those resources. Resources that can be placed into a namespace include network interfaces, the process ID list, mount points, IPC resources, and the system's own hostname information. As an example, two processes in two different mounted namespaces have different views of what the mounted root file system is. Each container is added to a specific set of namespaces, which are only used by that container.

***Control groups (cgroups)***

Control groups partition sets of processes and their children into groups in order to manage and limit the resources they consume. Control groups place restrictions on the amount of system resources the processes belonging to a specific container might use. This keeps one container from using too many resources on the container host.

***SELinux***

SELinux is a mandatory access control system that is used to protect containers from each other and to protect the container host from its own running containers. Standard SELinux type enforcement is used to protect the host system from running containers. Container processes run as a confined SELinux type that has limited access to host system resources. In addition, sVirt uses SELinux Multi-Category Security (MCS) to protect containers from each other. Each container's processes are placed in a unique category to isolate them from each other.

## Docker Container Images

In a nutshell, to create a new image, there are two approaches:

* Using a running container: An immutable image is used to start a new container instance and any changes or updates needed by this container are made to a read/write extra layer. **Docker** commands can be issued to store that read/write layer over the existing image to generate a new image. Due to its simplicity, this approach is the easiest way to create images, but it is not a recommended approach because the image size might become large due to unnecessary files, such as temporary files and logs.
* Using a Dockerfile: Alternatively, container images can be built from a base image using a set of steps called instructions. Each instruction creates a new layer on the image that is used to build the final container image. This is the suggested approach to building images, because it controls which files are added to each layer.