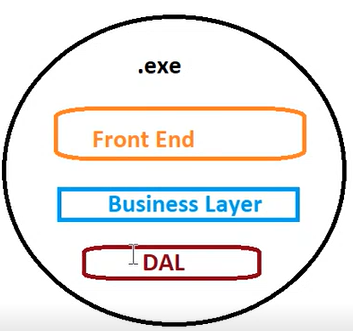
**Problems with Monolithic Architecture:**

1) Entire application is deployed as a single exe. Hence, if you make change in any layer (UI/BLL/DAL), then you need to redeploy the entire application.



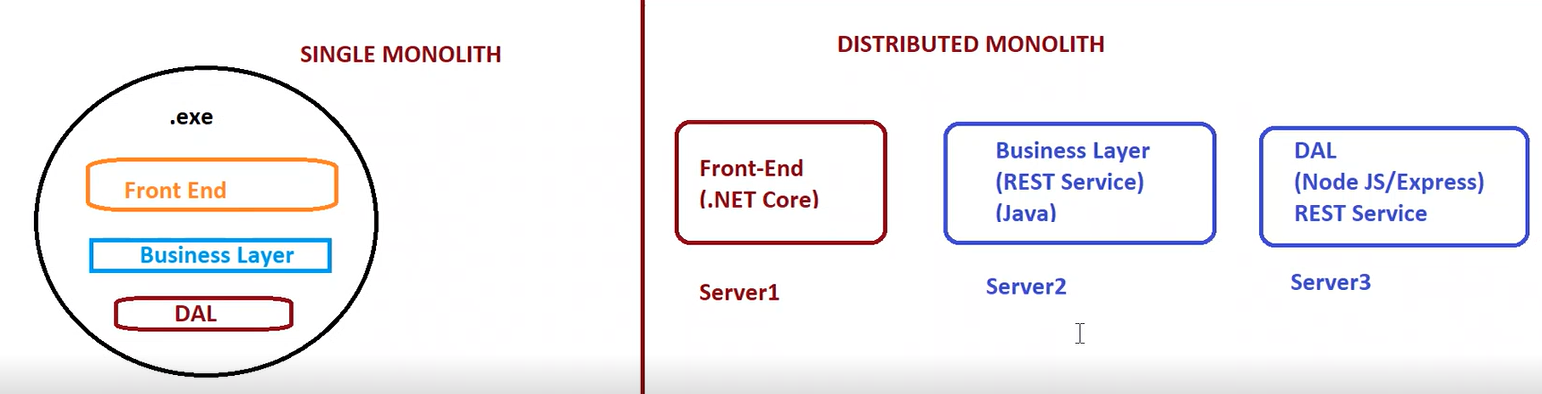
2) All the layers have to be coded using a single technology. All team members working on each layer must be aware of a technology used to code the project.

If someone doesn’t know the technology, then you may need to train the resource on that technology.

3) If you have developed application using some technology (E.g. - .NET), then environment where application is deployed should have NET. If you are deploying .NET web-based application, then deployment environment must have IIS installed. That means application has to be deployed on Windows as IIS is tied to Windows.

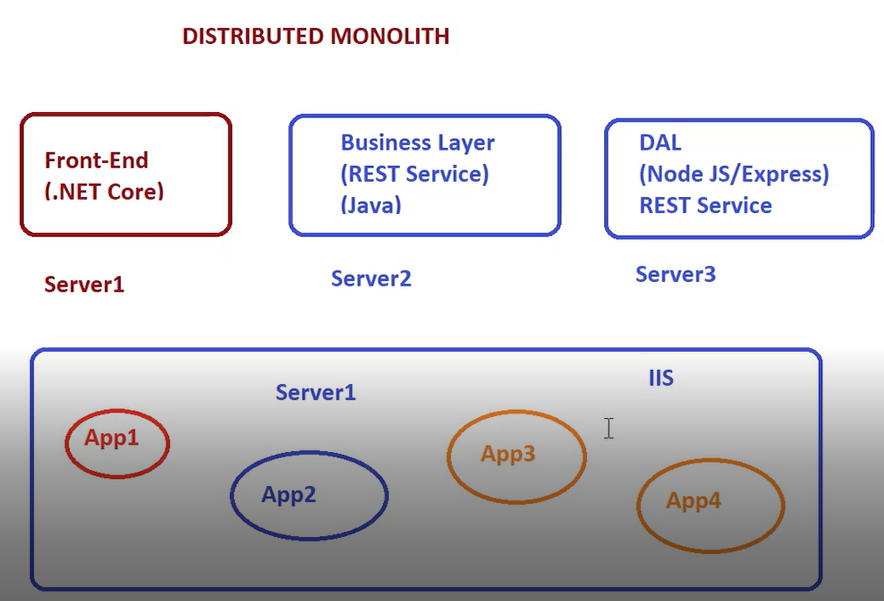
So, the problem is any minor changes made in the deployment environment will affect your exe.

To eliminate these problems, you had **distributed monolithic applications**.



This fixes the problems if Single Monolith. But the approach is still Monolithic approach.

**Problem with Distributed Monolith:**

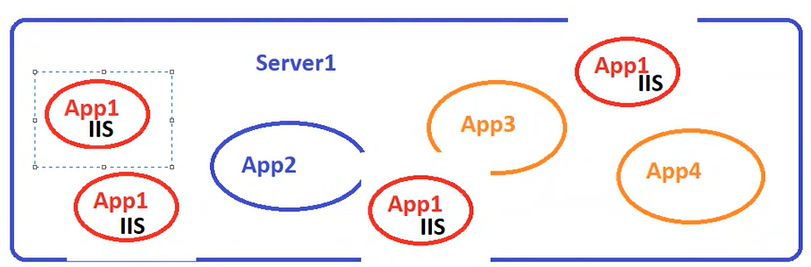


1) **All the applications are dependent on IIS.** Hence, if any configuration change is made in IIS, it will affect some applications for whom the change was not intended. Hence, those applications may need to be changed and redeployed again. Monolithic application is not self-dependent. It depends on some external environment. This problem persists even in distributed monolith.

2) Any change in external environment requires **application to be redeployed**.

3) **Difficult to Scale:** Difficult to create additional instances of a particular application on the same web server if required. There is no mechanism to recreate the application instances that have gone down. If you created replica of a server to scale a single application, then other applications which were not intended to be scaled are also scaled. Also creating a replica of a server is costly.

**Microservice Approach:**



You write a Dockerfile. In that Dockerfile, you specify a base image. Then you copy the build output into that container. Effectively, you are actually converting your application to Microservice only. So, when you start that container, container has got some entry point (specified using CMD). This entry point/CMD starts a web server. Hence, deployment environment is not expected to have web server installed.

**You can deploy Microservice on any environment which knows how to handle / manage the Microservices. Kubernetes is one such platform which knows how to handle / manage the Microservices. Docker Swarm is another platform which knows that.** These are all **Container Orchestration Platforms.**

Docker is all about deploying your applications.

Deployment approach is Microservices approach. Microservices are Not an API, Not a Programming Language. It’s a way of creating and deploying applications.

Microservice is any kind of application created using any programming language. The application does not handle multiple responsibilities.

Microservice can be any application handling a single responsibility. E.g. – Patient Management System performing CRUD operations on Patients can be a Microservice.

Microservices are Self Hosted. They don’t depend on any external environment to be deployed. E.g. – If your Microservice needs a Web Server to function, when you deploy a Microservice, it will be deployed along with the web server. When you start the Microservice, it will start its own web server and it will host itself. Hence you can put that Microservice on any environment. It will execute in the same way as it executed when you developed it. This is one of the major benefits of Microservices. In simple words, Microservice is a self-hosted application which runs inside its own process. It does not require any separate environment for execution or hosting. It comes with its own self hosted environment.

Microservices are:

1) Self-Hosted

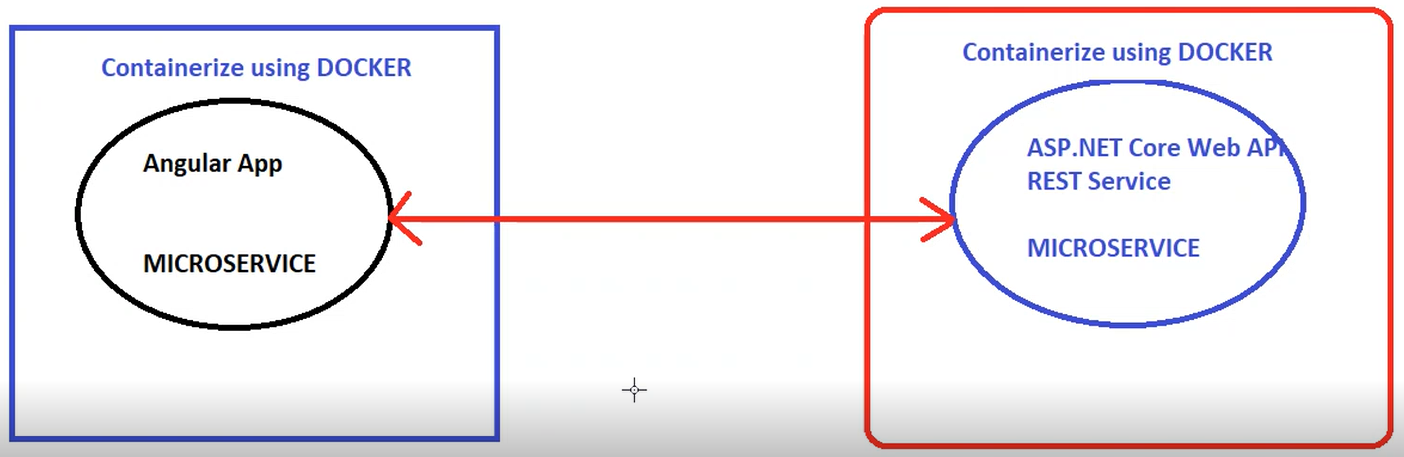
2) Can be Independently Deployed

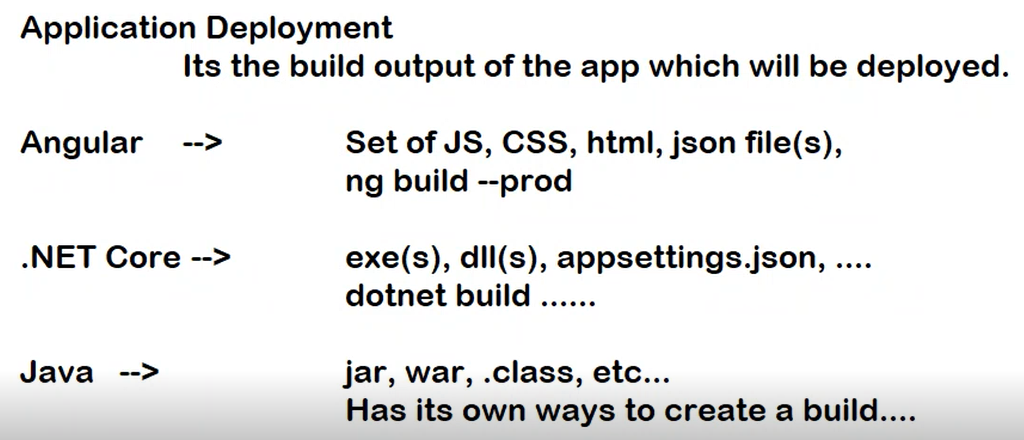
3) Can be Independently Versioned

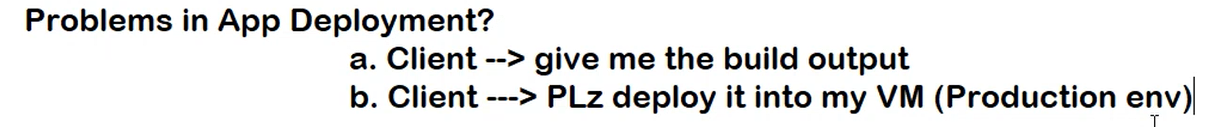
Docker plays a very important role when it comes to Microservices approach.

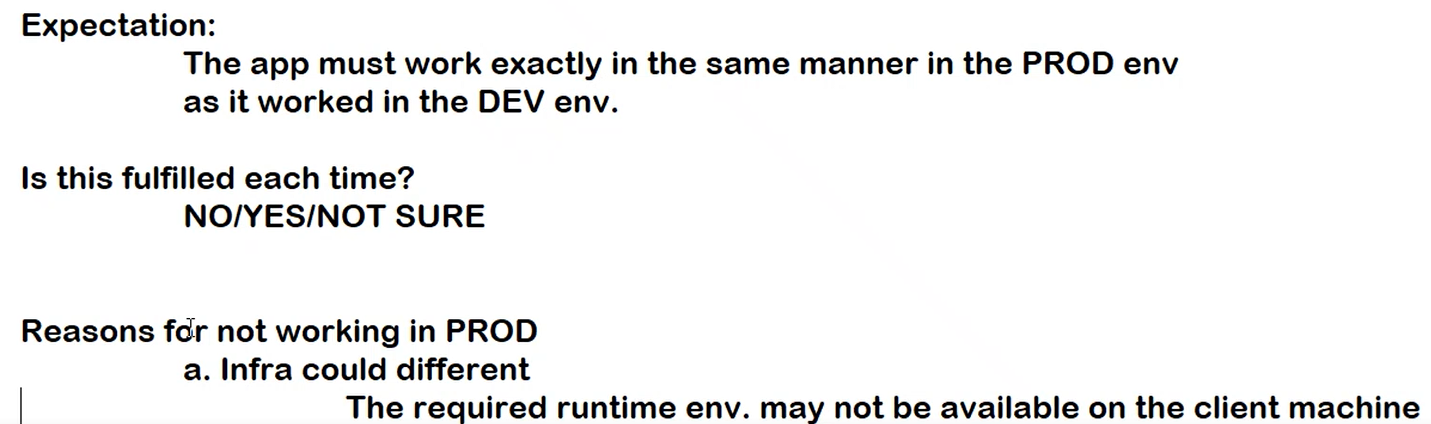
**Training Objective:**

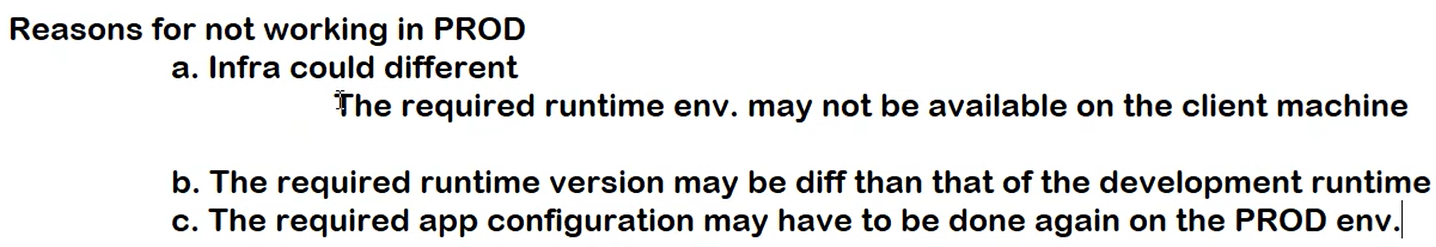








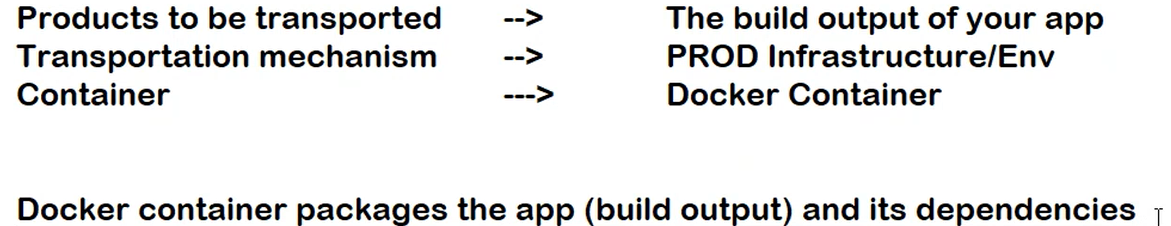






When you put the products inside the container, you package the products inside the container. You package it in such a way that they are protected. The packaging will be done in a standard way. Manner in which you package the products is the same irrespective of the mode of transport. Container solves this problem.

Without Containers, products will be packaged differently depending on a transportation medium.



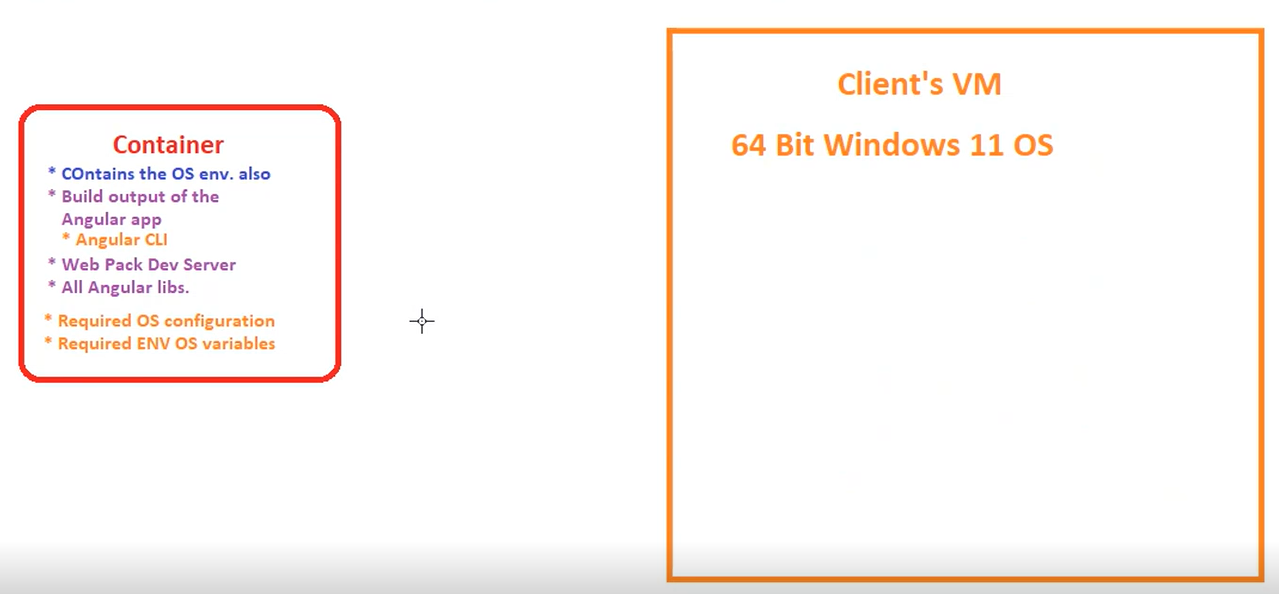


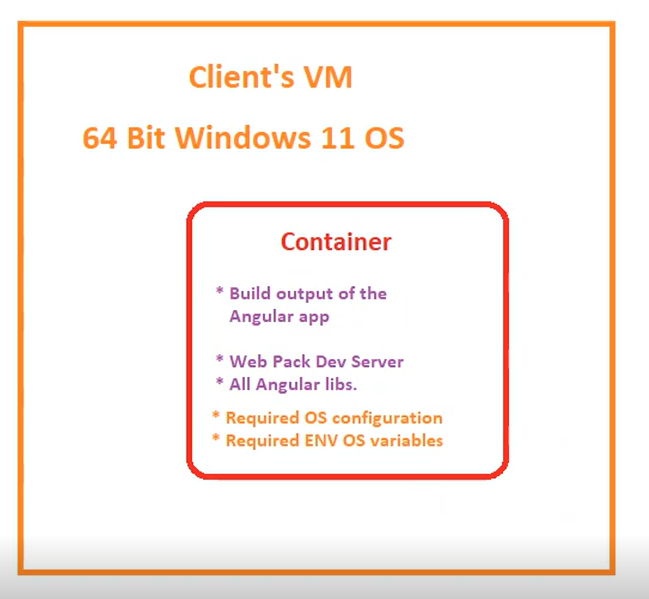
If you pick up the container and put it on Client’s VM, will your application work?



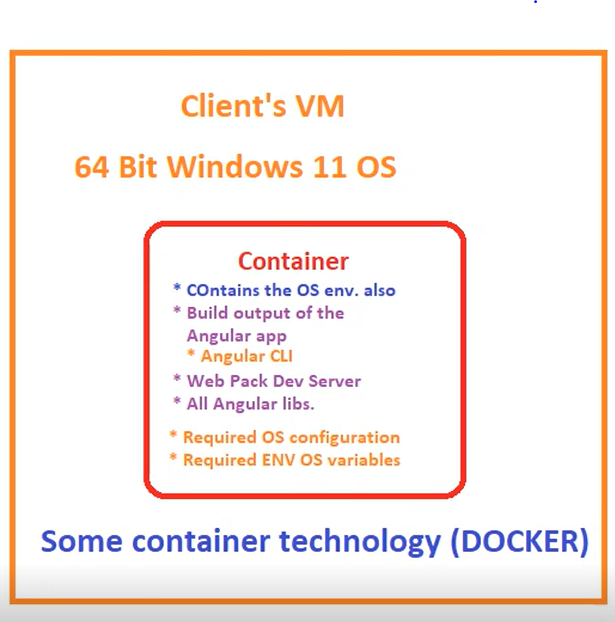
Will the app run when container is put on Windows 11 OS?

No.



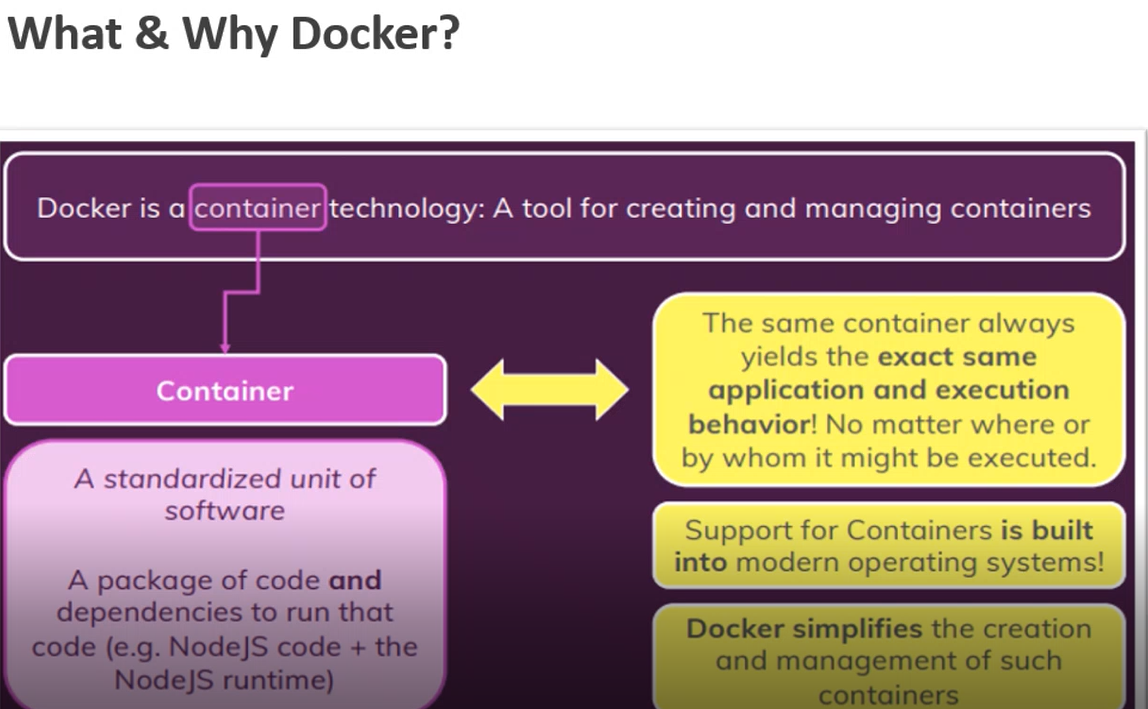


**Will the application work now? – YES. Only required thing is OS should have Docker installed. Because Docker understands how containers work.**



Docker as a tool simplifies creation and management of containers. You don’t need to learn that.

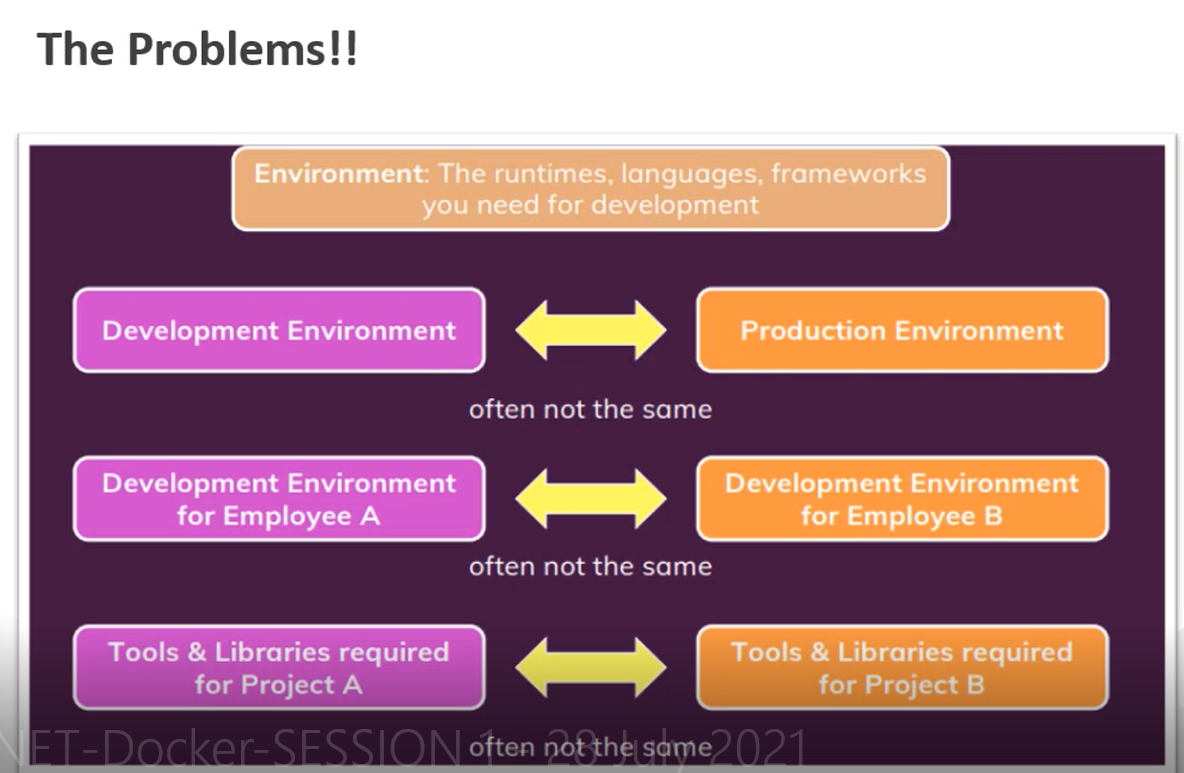
Docker is a container platform. It allows you to create and manage containers on any Operating System.



**Docker allows you to create and manage containers on any Operating System. Docker will take care of the things like How to create a container, How to start a container, How to stop a container, How to execute a container, How to destroy a container etc.**

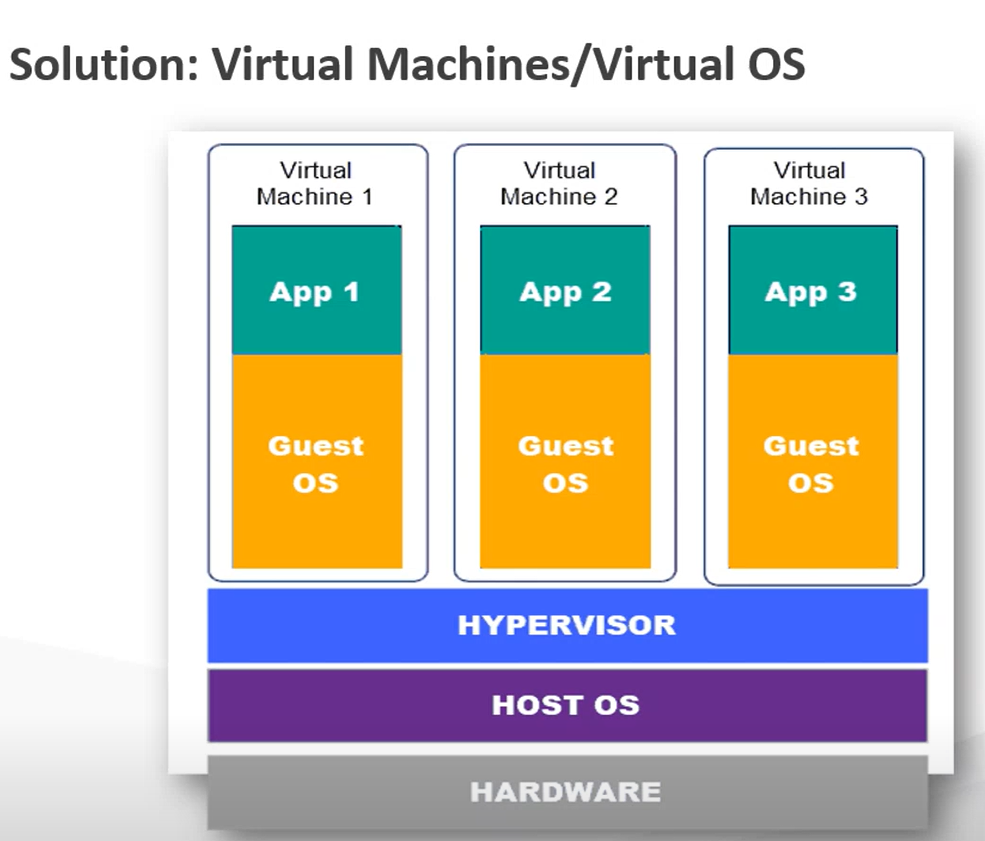
Though Operating Systems (OSs) can manage the containers, you don’t need to learn it as Docker will take care of it. Docker gives you the layer of abstraction over OS. Docker simplifies the creation and management of containers.







**VMs V/S Containers:**

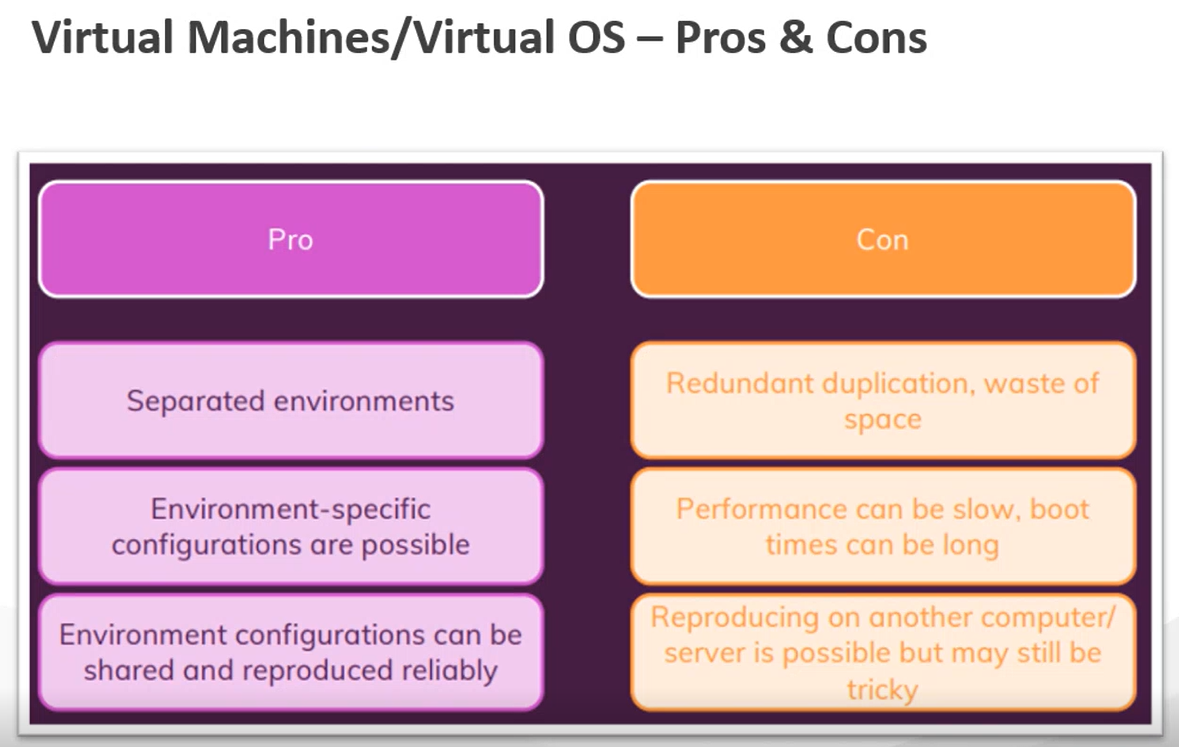


Hypervisor is a technology which allows Host OS to communicate with VMs and vice a versa. Hypervisor is a Microsoft technology.

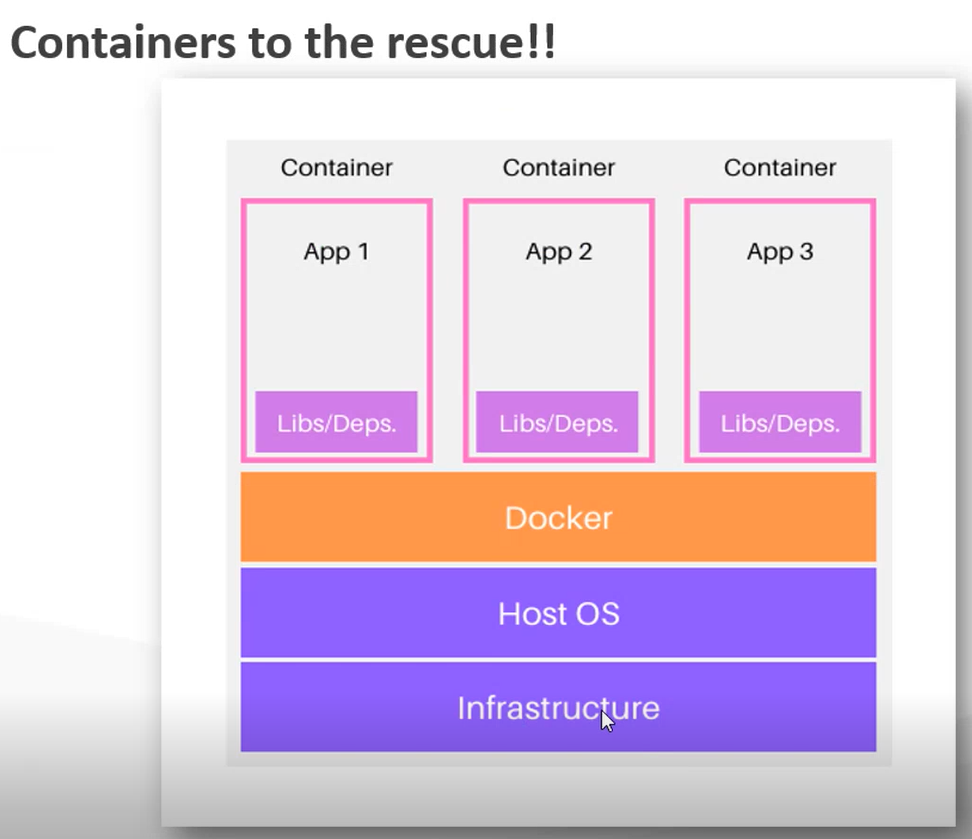
Each VM will have its own copy of an OS. Each of those OSs will have their own copy of the Kernel. Kernel is the piece of OS that talks to hardware.

**VM Advantages:**

You can configure the environment as per your requirement on VMs. **E.g. –** One VM having Development/One VM having Staging/One VM having Production environment.



**If the VM that uses Hypervisor needs to be created on another OS, that OS must have Hypervisor. If that OS has VMWare, then VM using Hypervisor cannot be recreated on that OS.**



**Docker will Create the Containers.**

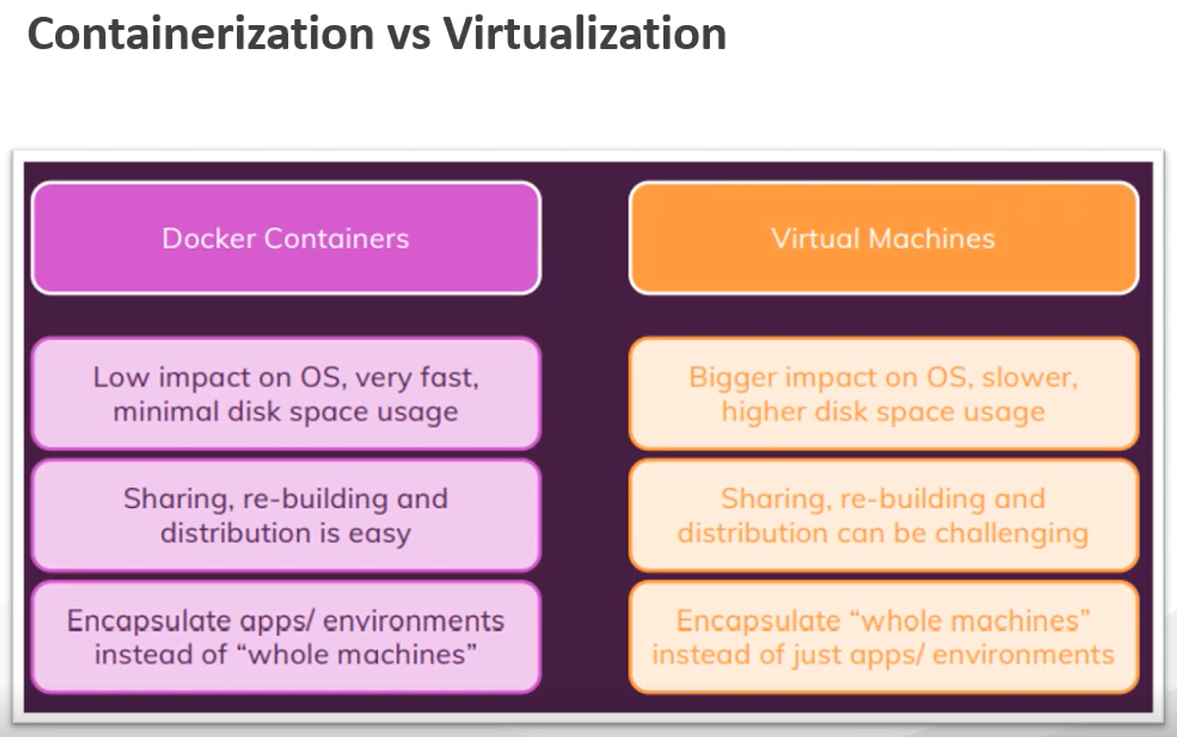
**Containers do not contain OS and Kernel. They contain only the app, its dependencies and basic file system of an OS.**

**They are very lightweight compared to VM image.**

**They can easily be started and stopped.**

**They share the Host Machine’s Kernel.**

**They can be easily migrated to any environment that has docker.**



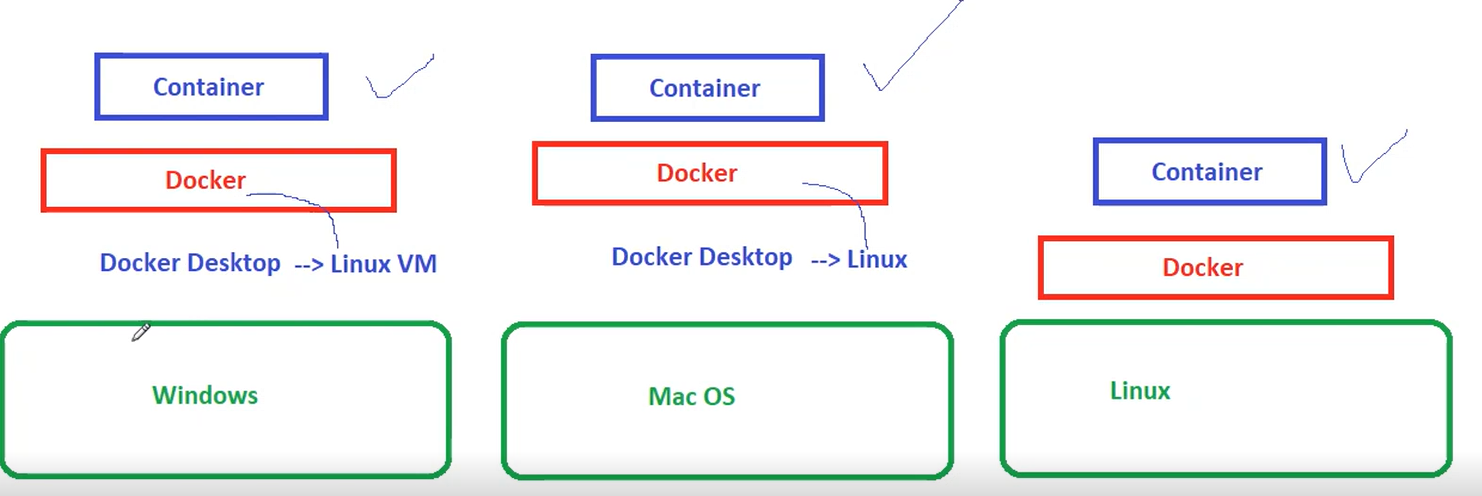
**Docker Installation Link:**

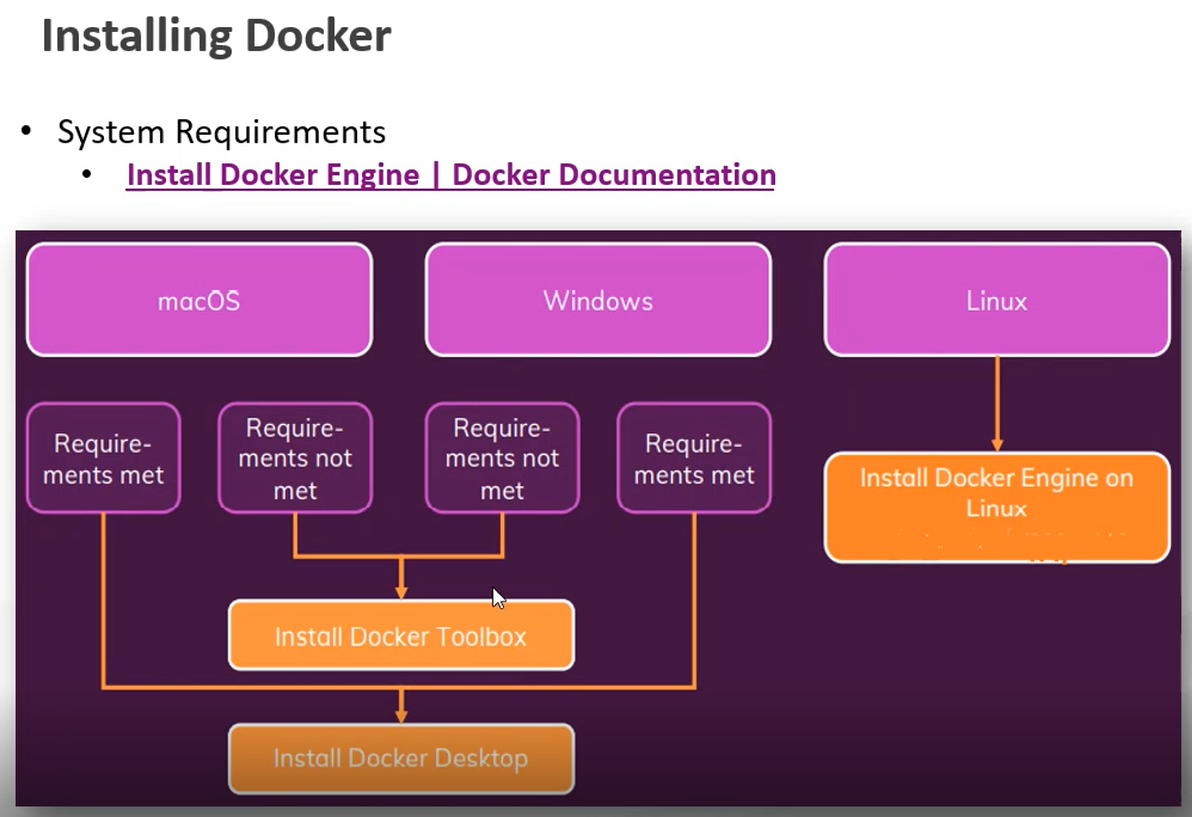


**Docker is written in Go language and it runs on natively on Linux. You can directly install Docker on Linux OS.**

**The Docker running on Linux will not run on Windows and Mac.**

**This can be done by installing Docker Desktop on Windows and Mac. Docker desktop internally creates a Linux VM and installs Docker on that VM.**





Docker Toolbox allows you to install Docker on older versions of OS.