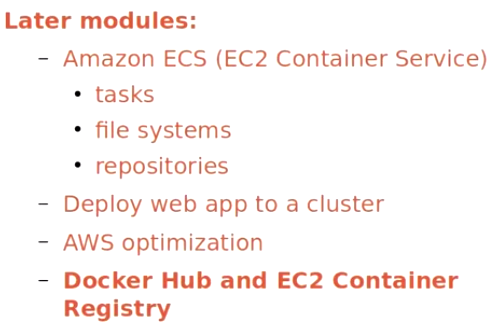
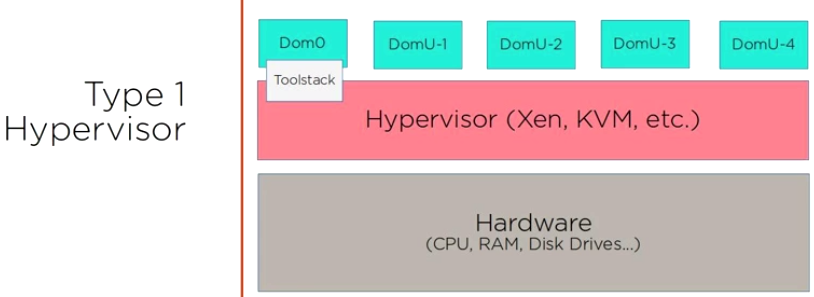
Using AWS, we can even host swarm of Docker containers

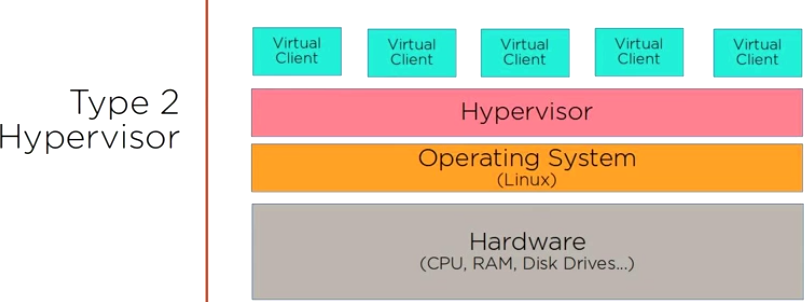


**What is Docker?** It’s a scriptable software container cunningly disguised as a server. And **what's a container?** Well, to start, it is not a hypervisor.

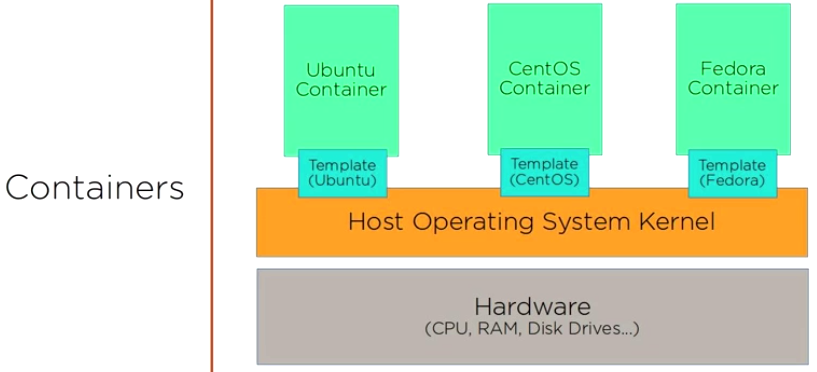
Let me explain that. As I described in much greater detail in my Linux Server Virtualization course, the compute, memory, storage, and network resources of a single physical computer can be divided up into an almost unlimited number of smaller, virtual computers.



Rather than having to go through the long and expensive process of purchasing and then building a physical server for every new need that comes up, these virtual devices can be quickly launched and shut down to fit constantly changing business needs. Since any particular virtual server might exist for a very short time - sometimes less than a minute - there is no need to over-provision to anticipate future needs. Instead, you give it exactly the resources it needs for the task at hand and, when it's done, you kill the server off to free its resources for whatever's coming next.

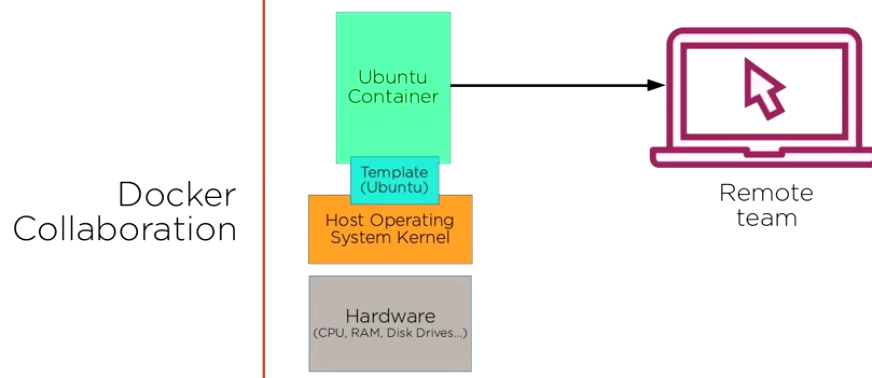
In fact, cloud-computing providers like AWS all use virtualized computers of one kind or another. The hundreds of thousands of Amazon EC2 instances all run on top of the open source Xen hypervisor - which is itself installed and running on many thousands of physical servers maintained in Amazon's vast - and secret - server farms. Type-1 hypervisors like Xen and VMware's ESXi are installed as a physical machine's operating system, providing each virtual guest operating system with seamless controlled access to shared hardware resources. Type-2 hypervisors - like VirtualBox - run on \*top\* of host operating systems, providing similar abstracted environments in which \*guest\* operating systems can imagine themselves running all by themselves on their own physical servers.

What both hypervisor types have in common, however, is that their job is to provide some kind of hosting environment for multiple complete, self-contained virtual computers.

Container systems like Docker, on the other hand, aren't standalone virtual machines, but are modified file systems sharing the operating system kernel of their physical host. While having to rely on an underlying kernel can be a bit restrictive,

* The huge advantage of containers is that they're each so lightweight: because they require so little space and system memory, many more of them will fit onto a given physical server than hypervisor guests.
* launching brand new containers can take just a few seconds
* **Moving containers around and sharing them between host systems is trivial..**.something my own painful personal experience can testify is often not true of the alternatives.

Now, for all their incredible features, you're probably not going to use Docker containers to watch streaming video of professional hockey games - although I have to admit that I'm really curious to know if such a thing is possible.

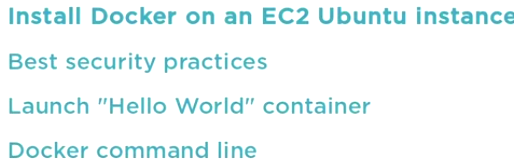
**Instead, you're far more likely to load up some kind of app development project to test how it will work, and then share it among your team members for feedback and updates. And once your app is complete, you might want to launch a cluster of containers (or "swarm" as Docker calls it) that can be programmatically and instantly scaled up or down according to user demand. **

As I said just before, there are all kinds of ways to implement Docker solutions within the AWS universe. 1.) AWS Elastic Beanstalk, which is a managed service that invisibly takes care of all the underlying infrastructure needs of an application, leaving you free to focus exclusively on application development. As you can see, available Beanstalk application environments include a number of Docker options.

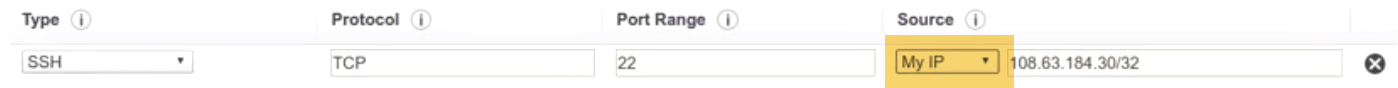
2.) AWS CloudFormation allows you to configure any combination of AWS resources into a template that can be deployed once or many times along with any specified dependencies and custom parameters. **Docker itself offers its Docker for AWS service (currently in beta), that will automatically generate a CloudFormation template to orchestrate a swarm of Docker containers to run on AWS infrastructure within your account.**

3.)Docker Datacenter for AWS is a joint AWS/Docker project that provides commercial customers with a more customizable interface for integrating Docker with CloudFormation.

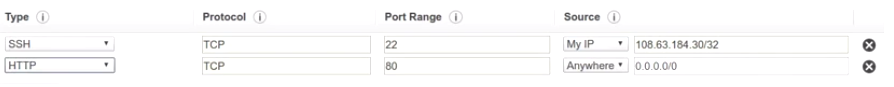
---------------------------------------------------------------------------------------------------------------------------------------







Restrict SSH to MyIP

ADD🡪http access – for web server...and a web server that can't be accessed by clients isn't much of a web server, is it?

Once the instance is running, I'll use my key to log in from my own local Linux workstation using SSH. This usage assumes that the plural.pem key is in the current directory. 

Docker requires a Linux kernel of 3.10 or above.

**how do you know that the installation package you're getting is actually coming from Docker?** Are you sure that you're not the victim of a redirect or man-in-the-middle attack that's delivering you malware.

So we'll definitely take advantage of Docker's encryption certificates by installing ca-certificates (assuming that it's not already there) and then adding the GPG key by copying the apt-key command from the Docker guide. Unless, that is, you're interested in typing that whole mess in manually. 

Next, we tell Docker the name of the container we'd like to launch, webserver in our case. And after that, we tell Docker to run a single command once the container is running to get the Apache webserver up. Some of you might wonder why I didn't use the more modern Systemd command, "systemctl start apache". Well I tried it, and discovered that, at this point at least, systemd is good and broken in Ubuntu Docker containers. Stay away if you know what's good for you. -D FOREGROUND ensures that Apache - and the container as a whole - will remain running even once the launch has completed. Let's run it.

We're given an ID for the new container, but nothing else. I'll run docker ps to list all running containers, and our webserver is listed.

I've installed curl on the Docker host, so I'll run it against localhost - meaning the EC2 instance we're logged into now - to confirm that the container's webserver is alive. It looks like everything's a go here.

Finally, I'll move to a browser on my local desktop and point it to the public IP address used by my AWS EC2 instance, which is, after all, our Docker host. And the page loaded successfully. We've got ourselves a living, breathing Docker web server container.

We've already enjoyed some of the benefits Docker Hub has to offer: the images we used to build the containers on the previous clips were all seamlessly downloaded from Docker Hub behind the scenes. In fact, using docker search, we can manually comb through the repository for publicly available images that are precise fits for our needs. We might, therefore, want to see if anyone has already adapted Ubuntu 16.04 for use as an Apache webserver - which could save us the trouble of building it ourselves. Running docker search apache/ubuntu shows us a rich variety of images from which to choose. The thing to keep in mind, is that there's no way to know how reliable any of these really is. As you can see, none of them has earned more than two stars, and none is designated as an "official" image.

By contrast, searching for ubuntu alone, returns very different results. Here we find official images with rather high star ratings, leaving us feeling quite a bit more comfortable checking them out.

I've already got the Ubuntu image, but if I were interested in using, say, ubuntu-upstart, I could grab it by simply running docker pull, followed by the image name.

\*Docker images\* will then show us our new image along with the rest of our local collection, and using docker run against the image name will launch a container and drop us into its shell.

As we will see in a later module, you can also push your own images to your own repository on Docker Hub and then, by making them public, allowing the entire world to pull them for their own purposes. If you decide to keep an image private, you can share it only among your own team members, much the way you might use a tool like GitHub. This is probably the most popular use-case for Docker, as it allows team members working remotely - or devs working in the same office who are just lazy - to get instant and reliable access to the exact environments being used at every stage of a project's progress.

Now, since we're on an AWS EC2 instace, all this technically qualifies as "running Docker on AWS". But doing it this way doesn't really take advantage of the vast range of AWS resources, nor is it all that efficient. That glory, we'll leave for the remaining modules of the course.

Let's review what we saw in this module. We talked about how container virtualization solutions like Docker are lighter and more versatile than hypervisors like Xen or VirtualBox. We briefly discussed the various ways of deploying Docker on AWS, including ECS, Elastic Beanstalk, CloudFormation, CloudFormation via Docker for AWS, and the enterprise service, Docker Datacenter for AWS.

We installed Docker on our EC2 instance making use of Docker's encrypted keys, added AUFS file handling functionality and then added the Docker package itself.

We then explored the Docker command line, launched a simple container based on the Ubuntu image, and learned how to manage Docker networking. We also spent some time building a container using a simple dockerfile, and then launching it as a fully-functioning web server. Finally, we dug into Docker Hub and discussed its value as a collaborative development tool.