deis.com

A Developer’s Journey into Linux Containers

Bob Reselman

I’ll let you in on a secret: all that DevOps cloud stuff that goes into

getting my applications into the world is still a bit of a mystery to me. But, over time I've come to realize that understanding the ins and outs of large scale machine provisioning and application deployment is important knowledge for a developer to have. It's akin to being a professional musician. Of course you need know how to play your instrument. But, if you don't understand how a recording studio works or how you fit into a symphony orchestra, you're going to have a hard time working in such environments.

In the world of software development getting your code into our very big world is just as important as making it. DevOps counts and it counts a lot.

So, in the spirit of bridging the gap between Dev and Ops I am going to present container technology to you from the ground up. Why containers? Because there is strong evidence to suggest that containers are the next step in machine abstraction: making a computer a place and no longer a thing. Understanding containers is a journey that we'll take together.

In this article I am going to cover the concepts behind containerization. I am going to cover how a container differs from a virtual machine. I am going to go into the logic behind containers construction as well as how containers fit into application architecture. I'll discussion how lightweight versions of the Linux operating system fits into the container ecosystem. I'll discuss using images to create reusable containers. Lastly I'll cover how clusters of containers allow your applications to scale quickly.

In later articles I'll show you the step by step process to containerize a sample application and how to create a host cluster for your application's containers. Also, I'll show you how to use a Deis to deploy the sample application to a VM on your local system as well as a variety of cloud providers.

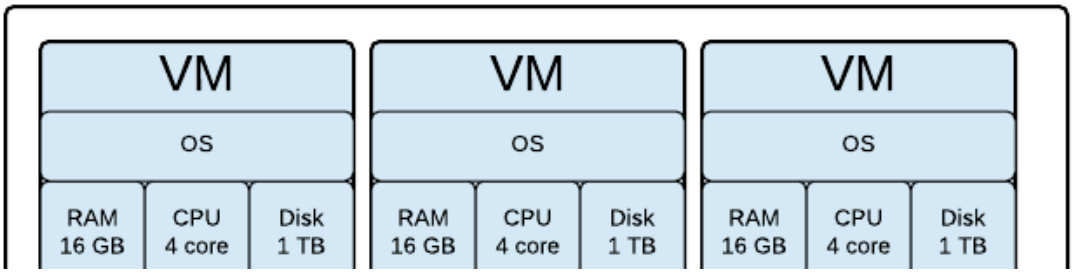
So let's get started.

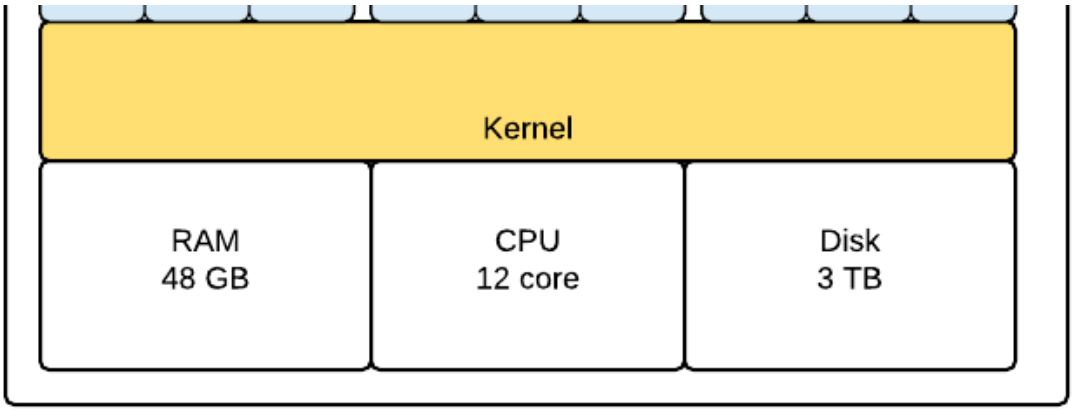
The Benefit of Virtual Machines

In order to understand how containers fit into the scheme of things you need to understand the predecessor to containers: virtual machines.

A *virtual machine* (VM) is a software abstraction of a computer that runs on a physical host computer. Configuring a virtual machine is akin to buying a typical computer: you define the number of CPUs you want along with desired RAM and disk storage capacity. Once the machine is configured, you load in the operating system and then any servers and applications you want the VM to support.

Virtual machines allow you to run many simulations of a computer on a single hardware host. Here's what that looks like with a handy diagram:





Virtual machines bring efficiency to your hardware investment. You can buy a big, honking machine and run a lots of VMs on it. You can have a database VM sitting with a bunch of VMs with identical versions of your custom app running as a cluster. You can get a lot of scalability out of a finite hardware resources. If you find that you need more VMs and your host hardware has the capacity, you add what you need. Or, if you don't need a VM, you simply bring the VM off line and delete the VM image.

The Limitations of Virtual Machines

But, virtual machines *do* have limits.

Say you create three VMs on a host as shown above. The host has 12 CPUs, 48 GB of RAM, and 3 TB of storage. Each VM is configured to have 4 CPUs, 16 GB of RAM and 1 TB of storage. So far, so good. The host has the capacity.

But there is a drawback. All the resources allocated to a particular machine are dedicated, no matter what. Each machine has been allocated 16 GB of RAM. However, if the first VM never uses more

than 1 GB of its RAM allocation, the remaining 15 GB just sit there unused. If the third VM uses only 100 GB of its 1 TB storage allocation, the remaining 900 GB is wasted space.

There is no leveling of resources. Each VM owns what it is given. So, in a way we're back to that time before virtual machines when we were paying a lot of good money for unused resources.

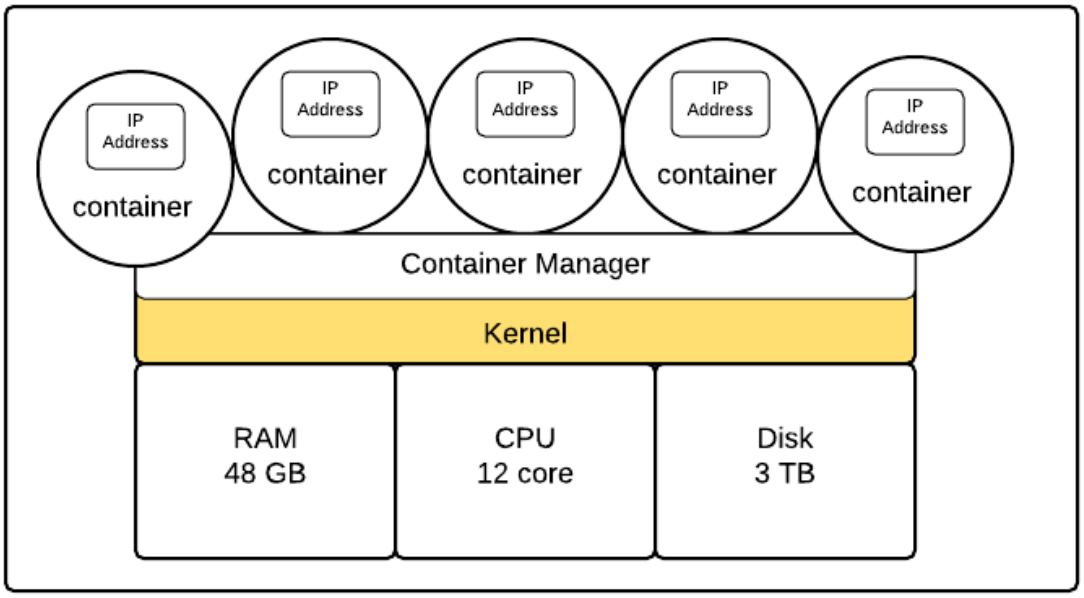
There is *another* drawback to VMs too. They can take a long time to spin up. So, if you are in a situation where your infrastructure needs to grow quickly, even in a situation when VM provisioning is automated, you can still find yourself twiddling your thumbs waiting for machines to come online.

Enter: Containers

Conceptually, a container is a Linux process that thinks it is the only process running. The process knows only about things it is told to know about. Also, in terms of containerization, the container process is assigned its own IP address. This is important, so I will say it again. *In terms of containerization, the container process is assigned its own IP address*. Once given an IP address, the process is an identifiable resource within the host network. Then, you can issue a command to the container manager to map the container's IP address to a IP address on the host that is accessible to the public. Once this mapping takes place, for all intents and purposes, a container is a distinct machine accessible on the network, similar in concept to a virtual machine.

Again, a container is an isolated Linux process that has a distinct IP

address thus making it identifiable on a network. Here's what that looks like as diagram:



A container/process shares resources on the host computer in a dynamic, cooperative manner. If the container needs only 1 GB of RAM, it uses only 1 GB. If it needs 4 GB, it uses 4 GB. It's the same with CPU utilization and storage. The allocation of CPU, memory and storage resources is dynamic, not static as is usual on a typical virtual machine. All of this resource sharing is managed by the container manager.

Lastly, containers boot very quickly.

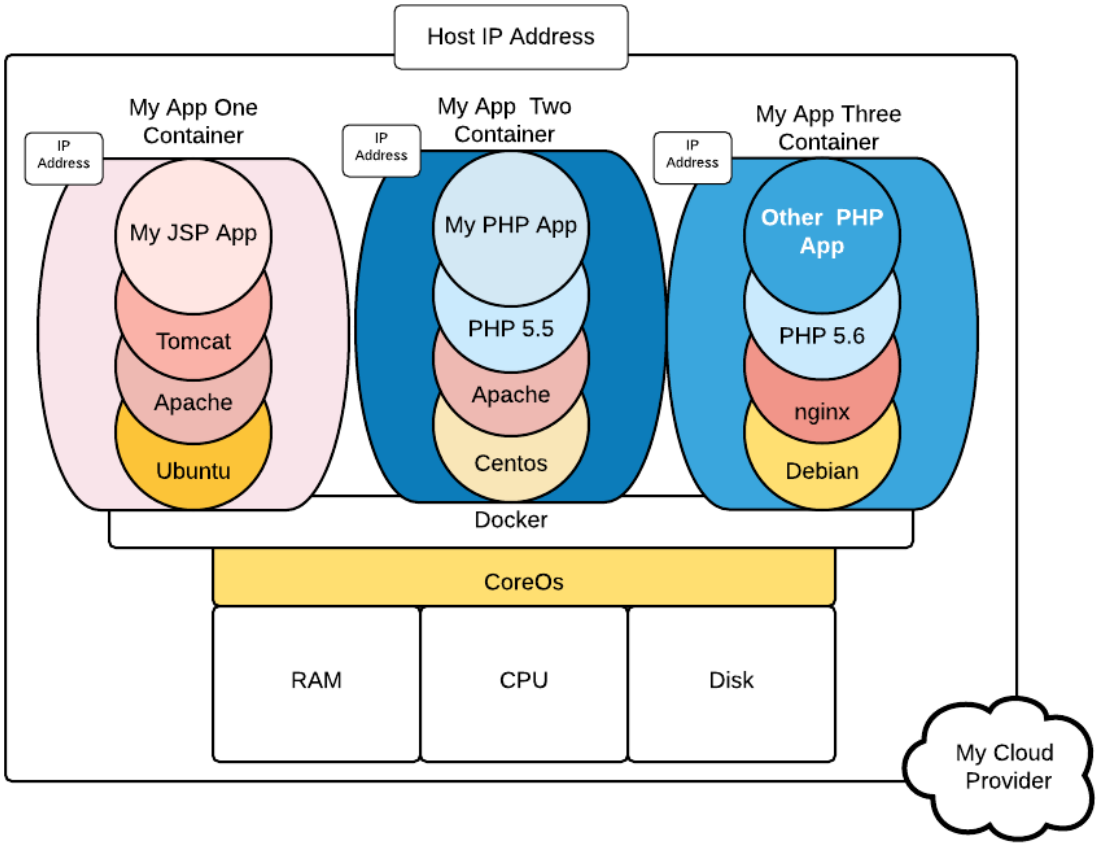
So, the benefit of containers is: *you get the isolation and encapsulation of a virtual machine without the drawback of dedicated static resources*. Also, because containers load into memory fast, you get better performance when it comes to scaling

many containers up.

Container Hosting, Configuration, and Management

Computers that host containers run a version of Linux that is stripped down to the essentials. These days, the more popular underlying operating system for a host computer is CoreOS, mentioned above. There are others, however, such as Red Hat Atomic Host and Ubuntu Snappy.

The Linux operating system is shared between all containers, minimising duplication and reducing the container footprint. Each container contains only what is unique to *that specific container*. Here's what that looks like in diagram form:



You configure your container with the components it requires. A container component is called a *layer*. A layer is a container image. (You'll read more about container images in the following section.). You start with a base layer which typically the type of operating system you want in your container. (The container manager will provides only the parts of your desired operating system that is not in the host OS) As you construct the configuration of your container, you'll add layers, say Apache if you want a web server, PHP or Python runtimes, if your container is running scripts.

Layering is very versatile. If you application or service container requires PHP 5.2, you configure that container accordingly. If you have another application or service that requires PHP 5.6, no problem. You configure that container to use PHP.5.6. Unlike VMs, where you need to go through a lot of provisioning and installation hocus pocus to change a version of a runtime dependency; with containers you just redefine the layer in the container configuration file.

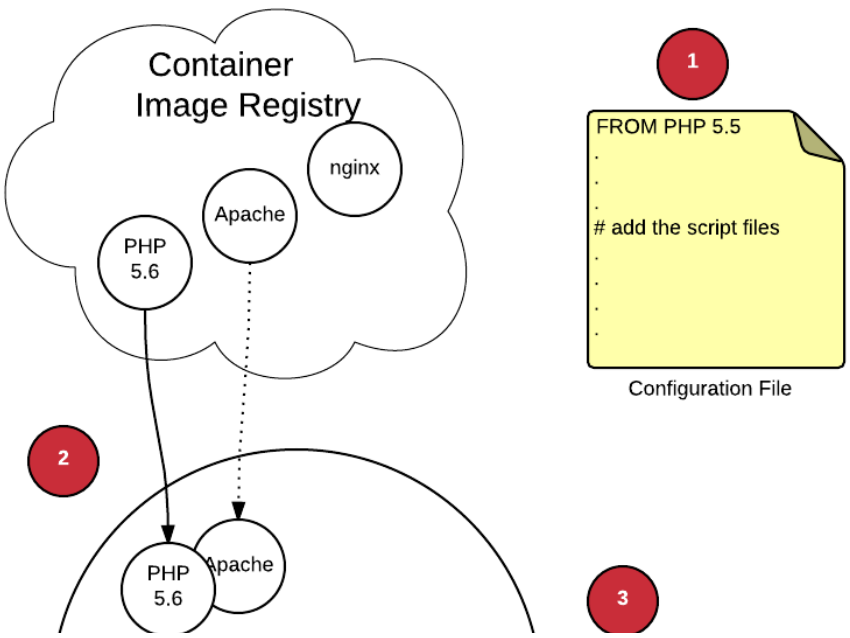
All of the container versatility described previously is controlled by the a piece of software called a *container manager*. Presently, the most popular container managers are Docker and Rocket. The figure above shows a host scenario is which Docker is the container manager and CoreOS is the host operating system.

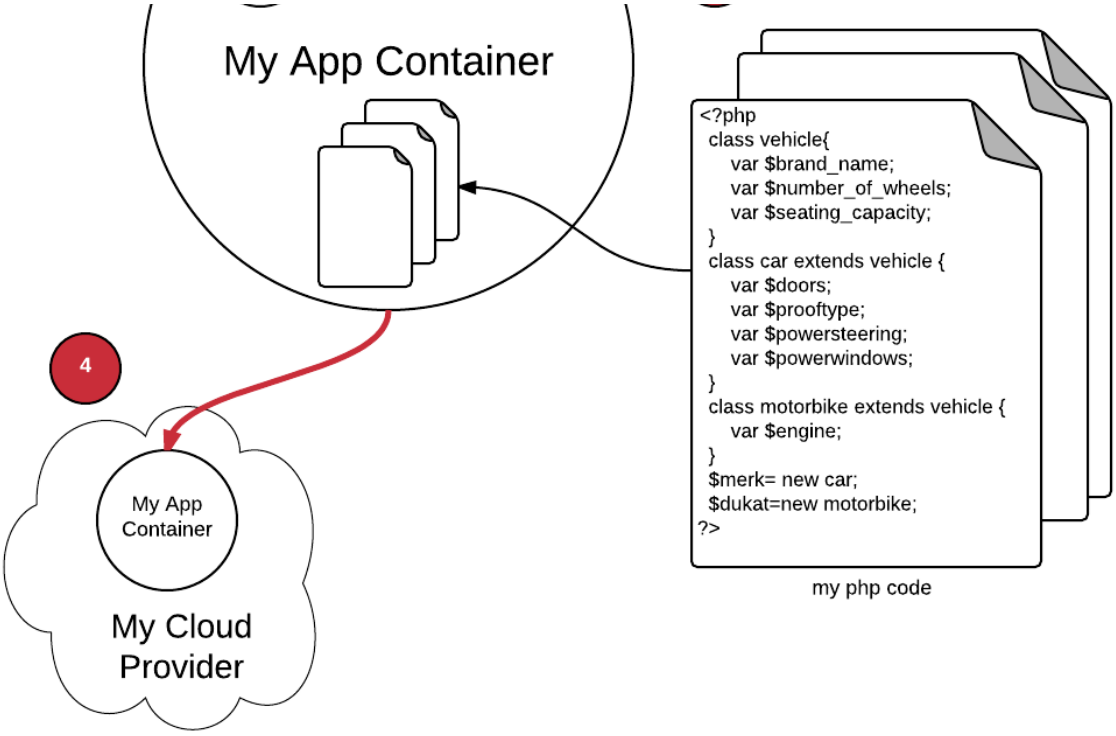
When it comes time for you to build our application into a container, you are going to assemble images. An image represents a template of a container that your container needs to do its work. (I know, containers within containers. Go figure.) Images are stored in a

registry. Registries live on the network.

Conceptually, a registry is similar to a Maven repository, for those of you from the Java world, or a NuGet server, for you .NET heads. You'll create a container configuration file that lists the images your application needs. The you'll use the container manager to make a container that includes your application's code as well as constituent resources downloaded from a container registry. For example, if your application is made up of some PHP files, your container configuration file will declare that you get the PHP runtime from a registry. Also, you'll use the container configuration file to declare the `.php` files to copy into the container's file system. The container manager encapsulates all your application stuff into a distinct container that you'll run on a host computer, under a container manager.

Here's a diagram that illustrates the concepts behind container creation:





Let's take a detailed look at this diagram.

Here, (1) indicates there is a container configuration file that defines the stuff your container needs, as well as how your container is to be constructed. When you run your container on the host, the container manager will read the configuration file to get the container images you need from a registry on the cloud (2) and add the images as layers in your container.

Also, if that constituent image requires other images, the container manager will get those images too and layer them in. At (3) the container manager will copy in files to your container as is required.

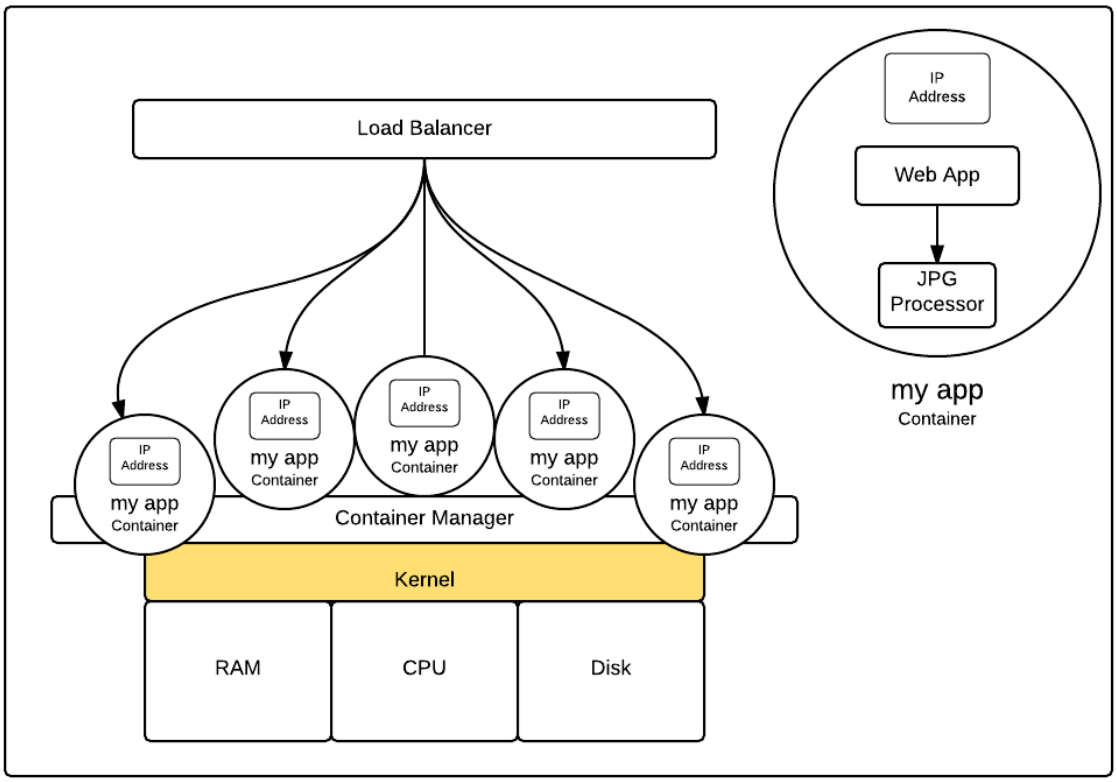
If you use a provisioning service, such as Deis, the application container you just created exists as an image (4) which the provisioning service will deploy to a cloud provider of your choice.

Examples of cloud providers are AWS and Rackspace.

Okay. So we can say there is a good case to be made that containers provide a greater degree of configuration flexibility and resource utilization than virtual machines. Still, this is not the all of it.

Where containers get really flexible is when they're clustered. Remember, a container has a distinct IP address. Thus, it can be put behind a load balancer. Once a container goes behind a load balancer, the game goes up a level.

You can run a cluster of containers behind a load balancer container to achieve high performance, high availability computing. Here's one example setup:



Let's say you've made an application that does some resource intensive work. Photograph processing, for example. Using a container provisioning technology such as Deis, you can create a container image that has your photo processing application configured with all the resources upon which your photo processing application depends. Then, you can deploy one or many instances of your container image to under a load balancer that reside on the host. Once the container image is made, you can keep it on the sidelines for introduction later on when the system becomes maxed out and more instances of your container are required in the cluster to meet the workload at hand.

There is more good news. You don't have manually configure the load balancer to accept your container image every time you add more instances into the environment. You can use service discovery technology to make it so that your container announces its availability to the balancer. Then, once informed, the balancer can start to route traffic to the new node.

Container technology picks up where the virtual machine has left off. Host operating systems such as CoreOS, RHEL Atomic, and Ubuntu's Snappy, in conjunction with container management technologies such as Docker and Rocket, are making containers more popular everyday.

While containers are becoming more prevalent, they do take a while to master. However, once you get the hang of them, you can use provisioning technologies such as Deis to make container creation and deployment easier.

Getting a conceptual understanding of containers is important as we move forward to actually doing some work with them. But, I imagine the concepts are hard to grasp without the actual hands-on experience to accompany the ideas in play. So, this is what we’ll do in the next segment of this series: make some containers.

Cartoon graphic provided courtesy of TheWorldInWhichWeLive.com