Docker originally used [LinuX Containers](http://lxc.sourceforge.net/) (LXC), but later switched to [runC](https://github.com/opencontainers/runc) (formerly known as **libcontainer**), which runs in the same operating system as its host. This allows it to share a lot of the host operating system resources. Also, it uses a layered filesystem ([AuFS](http://aufs.sourceforge.net/)) and manages networking.

AuFS is a layered file system, so you can have a read only part and a write part which are merged together. One could have the common parts of the operating system as read only (and shared amongst all of your containers) and then give each container its own mount for writing.

So, let's say you have a 1GB container image; if you wanted to use a Full VM, you would need to have 1GB times x number of VMs you want. With docker and AuFS you can share the bulk of the 1GB between all the containers and if you have 1000 containers you still might only have a little over 1GB of space for the containers OS (assuming they are all running the same OS image).

A full virtualized system gets its own set of resources allocated to it, and does minimal sharing. You get more isolation, but it is much heavier (requires more resources). With docker you get less isolation, but the containers are lightweight (require fewer resources). So you could easily run thousands of containers on a host, and it won't even blink. Try doing that with Xen, and unless you have a really big host, I don't think it is possible.

A full virtualized system usually takes minutes to start, whereas docker/LXC/runC containers take seconds, and often even less than a second.

There are pros and cons for each type of virtualized system. If you want full isolation with guaranteed resources, a full VM is the way to go. If you just want to isolate processes from each other and want to run a ton of them on a reasonably sized host, then docker/LXC/runC seems to be the way to go.

For more information, check out [this set of blog posts](http://web.archive.org/web/20150326185901/http:/blog.dotcloud.com/under-the-hood-linux-kernels-on-dotcloud-part) which do a good job of explaining how LXC works.

I feel foolish for asking, but why is deploying software to a docker image (if that's the right term) easier than simply deploying to a consistent production environment?

Deploying a consistent production environment is easier said than done. Even if you use tools like [chef](https://en.wikipedia.org/wiki/Chef_%28software%29) and [puppet](https://en.wikipedia.org/wiki/Puppet_%28software%29), there are always OS updates and other things that change between hosts and environments.

Docker gives you the ability to snapshot the OS into a shared image, and makes it easy to deploy on other docker hosts. Locally, dev, qa, prod, etc.: all the same image. Sure you can do this with other tools, but not nearly as easily or fast.

This is great for testing; lets say you have thousands of tests that need to connect to a database, and each test needs a pristine copy of the database and will make changes to the data. The classic approach to this is to reset the database after every test either with custom code or with tools like [Flyway](https://flywaydb.org/) - this can be very time consuming and means that tests must be run serially. However, with Docker you could create an image of your database and run up one instance per test, and then run all the tests in parallel since you know they will all be running against the same snapshot of the database. Since the tests are running in parallel and in docker containers they could run all on the same box at the same time and should finish much faster. Try doing that with a full VM.

**Edit:** From comments...

Interesting! I suppose I'm still confused by the notion of "snapshot[ting] the OS". How does one do that without, well, making an image of the OS?

Well, let's see if I can explain. You start with a base image, and then make your changes, and commit those changes using docker, and it creates an image. This image contains only the differences from the base. When you want to run your image, you also need the base, and it layers your image on top of the base using a layered file system: as mentioned above, Docker uses AUFS. AUFS merges the different layers together and you get what you want; you just need to run it. You can keep adding more and more images (layers) and it will continue to only save the diffs. Since docker typically builds on top of ready-made images from a [registry](https://registry.hub.docker.com/), you rarely have to "snapshot" the whole OS yourself.