For container images: hub.docker.com

To provision your vagrantfile with Docker: doc.docker.com

CONTAINERS

Containers are a form of operating system virtualization. A single container might be used to run anything from a small microservice or software process to a larger application. Inside a container are all the necessary executables, binary code, libraries, and configuration files.

Examples of container runtimes are:

* **runC,**
* **containerd,**
* **Docker,**
* **Windows Containers**.

There are three main types of container runtimes—

* low-level runtimes,
* high-level runtimes,
* sandboxed or virtualized runtimes

**What Is Container(ization)?**

Containerization is a process of packaging your application together with its dependencies into one package (a container). Such a package can then be run pretty much anywhere, no matter if it’s an on-premises server, a virtual machine in the cloud, or a developer’s laptop. By abstracting the infrastructure, containerization allows you to make your application truly portable and flexible.

## [Application Isolation](https://devopsbootcamp.osuosl.org/application-isolation.html#id6)

Application isolation is the separation of one program or application stack from the rest of the running processes. The oldest way to do this is to simply run your application on a separate computer, but that gets very expensive very quickly.

There are two main ways to tackle Application Isolation on one computer: Virtual Machines and Containers. They both achieve similar end results but achieve that end in different ways and offer different advantages / disadvantages.

### Containers vs. Virtual Machines

This all might sound like a concept of a virtual machine deployed from a prebuilt image, but containers work much differently. The main difference between a VM and a container is a container virtualizes the operating system, whereas a virtual machine is an abstraction of physical hardware. This difference results in containers being more efficient and portable than VMs.

So, how do containers achieve isolation and portability? In the case of a Linux system, the answer is cgroups and namespace isolation. You can run containers on Windows, too. The Windows kernel uses different mechanisms to achieve the same results. Instead of cgroups it has [**job objects**](https://docs.microsoft.com/en-us/windows/win32/procthread/job-objects), and instead of namespaces it has server silo objects. These features of Windows and Linux kernels allow building containers, which encapsulate everything your application may need without creating any conflicts with the host operating system and other containers.

Because they achieve the same functionality as VMs (isolation, ability to package all you need into one executable piece), some people call containers “lightweight VMs.” “Lightweight” comes from the fact that whereas a VM includes a full copy of an operating system, the application, and binaries and libraries that can take up space and be slow to boot. Containers share the operating systems and infrastructure with other containers, with each container running as isolated processes in user space. Since containers are fundamentally different than virtual machines in how they’re constructed, it’s best to avoid calling them “lightweight VMs.”

## Containerization vs. DevOps

When you talk about containers, you’ll often hear about DevOps too. We need to understand why that is. Containers are a technology, while DevOps is a set of practices, culture, and principles. The reason you often see them together is containers as a technology make implementing DevOps easier. We’ll explain why in a second, but it’s important to understand that they can exist separately. You can have containers without DevOps, and vice versa.

The thing to remember is one without another would be more difficult and less efficient. Containers are a natural fit for DevOps. There are a few reasons for that, but the main point is DevOps provides faster software delivery through closer cooperation between developers and operation teams, giving more freedom to developers and a “fail fast” approach.

### Let’s Talk About the Benefits

Containers help with everything. Thanks to containers, different environments (e.g., development, test, production) can be the same since you no longer rely on operation teams to make sure different servers are running the same software versions. What’s more, applications will be in the same “environment” (container), even on a developer’s laptop. You simply deploy the same containers on different environments. This removes the common problem of “it works on my machine but not on the test server.”

### Continuous Deployments

Continuous deployment becomes easier with containers too. That’s because containers in general are small (or at least they should be), so it just takes seconds to deploy a new version of a container. Also, if you’re using containers you probably architected your application as microservices. This means you can update different parts of your application independently.

### Flexibility

Another benefit of containers is different parts of your application can be written in different languages. Therefore, developers aren’t limited to one programming language but can use languages they’re the most comfortable with. This contributes to DevOps because it gives you more freedom in arranging teams.

### Fail Fast

When it comes to the “fail fast” approach, containers limit the scope of application code developers need to understand. To fix a bug, a developer (in most cases) only needs to understand how one container works, not the whole application (unless, of course, the issue spans many containers). So, it’s usually far easier to narrow down the potential issues and find the root cause.

And once it’s fixed, you can quickly deploy a new version of one container, and you’re done. No need for multiple teams to align to find the issue, or for an end-to-end test of entire application when a change to single piece of the application was made. No need for approvals and alignment across multiple departments to redeploy the whole application, either.

### The Downside

With all the benefits of containers comes a downside: the cost. Networking becomes much more complicated since containers need to talk to each other. This is usually done using REST API. Therefore, instead of having only front-end to back-end and back-end to database connectivity, you’ll have dozens of connections, creating a complicated networking mesh.

The same applies to logging. You’ll no longer have one place to read logs from; each container will create its own logs. You’ll have to aggregate them, and it might become more difficult to have a general overview of the whole application. There are, however, tools like SolarWinds® Papertrail™ to help you with that. **[Papertrail](https://www.papertrail.com/)** can aggregate the logs from containers and create a centralized, easy-to-understand overview of the application state. Managing container logs with Papertrail will allow you to enjoy the benefits of containerization while still maintaining the ability to quickly identify and troubleshooting issues.

For container images: hub.docker.com

To provision your vagrantfile with Docker: doc.docker.com