SHARE
LAST UPDATED ON JULY 29, 2019
PLUTORA BLOG - DEVOPS, SOFTWARE DEVELOPMENT, TEST ENVIRONMENT MANAGEMENT

Containerization: A Definition and Best Practices Guide

Reading time 13 minutes

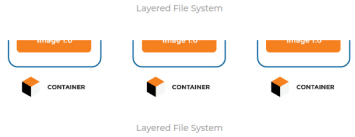
There are many ways for operations teams to approach hardware management. Some build dedicated hardware for different applications within their stack. Some use virtualization to simplify the task or manage dozens of applications. Both of these approaches come with serious drawbacks. Dedicated hardware is costly, both for the hardware and the power needed to run it. What's more, space is at a premium in many data centers. Adding a new server to an environment which is already squeezed for space is like the world's worst Tetris game.

Likewise, virtualization comes along with some serious problems. Virtual machines come with a whole host of issues in terms of dedicated resource management. They simplify the task of working in ever-changing data centers, but can be difficult to scale. **Being able to run four virtual machines on a single piece of dedicated hardware is a boon for space and energy-starved operations teams.** But it still means that you're devoting time and energy to maintaining redundant operating systems.

In the last five or so years, teams have been solving these problems by adopting containerization. If you're curious about containerization and what it can do for your organization, read on. **We're going to dive into just how it works, and some best practices that'll unlock the true power of containers** for your ops organization.

What Is Containerization?

In a lot of ways, containerization is best thought of as the natural evolution of virtualization. Virtualization treats each virtual machine as its own logically (but not physically) distinct server. Containerization treats each application as its own logically distinct server. Multiple applications will share one underlying operating system. Those containers don't know that any other containers are running on their dedicated hardware. If they want to communicate with another server, they need to communicate via a network interface, just like if they were on different physical devices.



The benefit is that you don't need to devote hardware resources to redundant operations. Instead of needing to dedicate CPU cores and memory to an operating system for each virtual machine, you build a server with one underlying virtual machine, and only dedicate cores and memory for the single application that runs in each container.

How Does Containerization Work?

We've hinted at this, but application containers work by virtualizing the operating system – specifically the Linux kernel. Each application container thinks that it's the only application running on a machine. Each container is defined as a single running application and a set of supporting libraries. **One significant way that containers differ from virtual machines is that containers are immutable.** That is, they can't be modified by any action that you take. This is on purpose—it means that every time you create a container, it'll be the same.

No matter where you create it, on what hardware or underlying operating system, the container will work exactly the same. That kind of consistency eliminates an entire class of bugs. If you're used to virtual machine snapshots, it's easy to think of a container in that way. Each time you start a container, it's restored to the initial snapshot no matter what actions you took the last time it ran.

That approach comes with some drawbacks, but it also packs a real punch

for an ops team. It means that each container will run exactly the same, no matter where you set it up. **A container running on a developer's laptop will run exactly the same as that container in your data center. This consistency eliminates an entire class of development and deployment bugs.** Your team never has to worry about some developer writing code against a different version of PostgreSQL or Java than your application uses in production. Your environment is identical everywhere it runs.

How Does Docker Fit In?

Docker is the most popular container system in the world. They're not the first library to support containerization, and they're not the only game in town, but they are the biggest. Right now, Docker powers somewhere between 75 and 80% of all production containers. To many ops teams, Docker and containerization are synonymous. As such, this article will be talking about some Docker-specific features, like Docker hub. That doesn't necessarily mean that Docker is the best solution, nor the best solution for your team's needs. It's simply the most popular example.

As with any other piece of software or hardware, diverging from the beaten path comes with positives and negatives. If you're considering embarking on a journey of containerization for your team, you should
talking about some Docker-specific features, like Docker hub. That doesn't necessarily mean that Docker is the best solution, nor the best solution for your team's needs. It's simply the most popular example.

As with any other piece of software or hardware, diverging from the beaten path comes with positives and negatives. If you're considering embarking on a journey of containerization for your team, you should spend time evaluating all the options on the market before choosing which makes the most sense.

What Are Some Best Practices for Working With Containers?

If you're working with containers, there are some best practices you need to know. Let's discuss a few of them.

Each Container Should Have Just One Application

This was difficult for me to wrap my head around when I first started building containers. I wanted to treat a container like a virtual machine. That meant stuffing it to the gills with all the different applications I needed to support my code. I'd drop a database and a message queue and a programming language run time all on one container.

This was, in short, a big mistake. I wasn't gaining any benefits of containerization. **For instance, when containers are well optimized, you don't ever have to deal with library compatibility issues.** It's a common issue to need to run one version of a code library to support your database, and another to support your message queue. In traditional hardware ecosystems, or even in virtual environments, you'll need to do complicated gymnastics to keep both versions of the library in usable locations.

With containers, you never need to worry about that. Your database container, which runs only your database, can run version 2.2 of that thorny library. Your message queue, which needs version 2.5? That's what it gets. They're running on the same underlying hardware, and even the same operating system, but they don't ever touch the same libraries. This feature saves hours of ops team stress and debugging in my experience.

Optimize Your Builds for the Build Cache

When you spin up a container, your container engine walks through each step in the definition and builds a container "layer." That layer is essentially its own snapshot of a container. **What's key is, every container with the same layers is identical.** So, for instance, a container that uses Debian as an underlying operating system and installs Ruby 2.5.4 is exactly the same, if those are the only two actions the container takes. As you design your container definitions, you can use that to your advantage. The reason why is because most container libraries (and Docker especially), cache their builds at each layer.

Since building a new batch of containers can be a time consuming process, a savvy administrator leverages the build cache to shorten those build times. In our hypothetical before, we were building containers to host a database server and a message queue. If both of those servers, for instance, ran on a Debian kernel and required Ruby 2.5.4, you would decrease your build times by structuring their build definitions similarly. We'd want to specify the requirements in each Dockerfile in the same order.

A Practical Example

This isn't valid Dockerfile syntax, I've simplified it to show the point

```
From Debian;

Use Ruby 2.5.4;

Install MessageQueueApplication;
and
```

```
From Debian;

Use Ruby 2.5.4;

Install DatabaseApplication;
instead of
From Debian;

Install MessageQueueApplication;

Use Ruby 2.5.4;
and
From Debian;

Install DatabaseApplication;

Use Ruby 2.5.4;
```

The reason for this is because the order of commands in the container definition matter. In the first two instances, we can use a cached version of the container that utilizes Debian + Ruby 2.5.4.

For one or two containers, this isn't a big deal. Especially because Docker caches each step of the build process. So, if your container definition doesn't change, once you've built a container once, it'll be cached on your machine. You don't have to worry about build times on subsequent runs. **But if you're managing a swarm of containers with dozens of individual instances, this can be a real time saver.** That's especially true if you're utilizing a continuous integration system that rebuilds containers whenever developers push code. It's a simple optimization that will save your organization significant time.

Build the Smallest Image Possible

This is another way of "hacking" the Docker container process to simplify your build steps. Each command you issue to the Docker container adds another layer (like in the previous section) to the Docker cache, and to your container. So for instance, running **apt-get update** as one command, then running **apt-get install some_package** as a separate line on your docker file means that you've just created two different layers. That's the kind of thing that will come back to bite you in overall container size. Instead, a savvy administrator will look for Docker build steps that can be combined with one another.

The size savings for this kind of optimization can be significant—dozens of megabytes per container. While storage is cheap, I'd imagine it's not so cheap that you're not willing to save dozens of megabytes on every single container you build and cache for a few minutes of optimization time.

Go Distroless to Save Space

Another way that you can streamline the size of your containers is by using so-called "distroless" images. Most ops teams don't set up every individual Docker container from scratch. Instead, they use pre-bundled containers as a base, then build on top of those. **This makes spinning up a new application container simple and painless,** but it has a real cost associated. For instance, in our previous example, we were starting with the Debian container image. This is a base that contains all the things you'd expect to find in a normal Debian installation.

Unfortunately, that includes things like software updating mechanisms and the bash shell. For an application container that's only ever running a message queue, all of those included applications are useless. Thankfully, those "distroless" applications strip out things that aren't necessary for running a basic application. These images can be a bit more difficult to work with, so you might not want to use them when you're first spinning up an application. But when you're deploying to production, a distroless container image can reduce the size of your container by 90%.

Take Advantage of Docker Hub or a Registry

A clever reader likely noticed that we've been talking a lot about images that other people built before us. Our examples of distroless images or the Debian image are examples of pre-built containers. **We can use these containers as building blocks, and add new layers on top of them.** That simplifies getting started with Docker to the point where anyone with a text editor and a command line can do so in a few hours.

But we can take our usage of Docker to the next level by plugging our containers into a centralized repository. Instead of simply sending out Dockerfiles which contain instructions on how to build our containers, we can build them once, and upload them to a storage server. Now, when we're deploying a new container, we can tell the Docker code that we want to download a specific image from a specific place. For the public, this is Docker Hub. That said, if you're someone who wants to keep your images a bit more under wraps, it's possible to host your own Docker Registry.

Whichever way you decide to go, centrally organizing your images is a great way to save time and headaches. You want your build process to be the same process that your developers use to set up their local systems. By moving to a centralized repository, you make sure your team is able to review all build changes before they go live.

Tag Your Images

If you do use a centralized repository, this step is a must. Like we've noted, Docker builds can add up quickly. Containers are just like any other piece of software. They don't stand still. You'll be constantly updating the libraries and applications that run on those servers to take advantage of new features and security updates. For most environments, those kinds of changes mean extensive testing of the new software to ensure that everything still works. Sometimes, you might wind up with several iterations of testing as your team irons out bugs.

That's where container tagging comes into play. While you're still testing

you can't deploy a container to production. You don't know whether or not it'll break something. But you still need to deploy that container, just to your testing environment. The solution is to version your containers. This is another real advantage that containers hold over virtual machines. **You can have your developers developing against a container with a new feature, while QA tests a security update.** Meanwhile, production is still running a known-good container. Once QA approves that security update, you update the production environment by updating the version of the container that's running there. QA can start testing the code with the new feature by updating the version of the container in the test environment.

Each step of the way, all your team needs to do is ensure the container builds correctly once. From there, it's uploaded to a central repository, tagged with a version, and each environment downloads it for their own use.

Containerization Unlocks Flexibility

The modern ops team needs to run more applications, at a larger scale, than ever before. Often, they're also doing this with fewer people than they've had in the past. Thankfully, modern tool sets unlock serious multipliers on administrator effort. Containerization has the capability to do that for any team. It erases whole classes of support issues, and simplifies a lot of deployment situations. This article didn't even get into the ability to use software to horizontally scale your container applications or how you can partner with a company like Plutora to build a release management stream.

Containerization unlocks huge potential for your team. **By leveraging these tools, you can not only build more robust applications, but you can do them in less time, with fewer people.** Instead of constantly trying to play catch up, you and your team can build robust environments that are ready for your constantly changing business. It's not the right answer for every business, but for teams where it works, it's a great advancement in technology. Is it right for your business?

Eric Boersma

Eric is a software developer and development manager who's done everything from IT security in pharmaceuticals to writing intelligence software for the US government to building international development teams for non-profits. He loves to talk about the things he's learned along the way, and he enjoys listening to and learning from others as well.

Readers also check out

Blue-Green Deployments: What They Are and How to Use Them

What Is a Steering Committee and What Is Its Role?

AIOps: What It Is and Why It Should Matter to You

The Critical Role of Value Stream Management in the Future of Software Delivery