CONTAINERS & CONTAINERIZATION: A BEGINNER'S GUIDE



Building containers is necessary if you wish to start using Kubernetes (K8s) or similar options. The only way to use Kubernetes, which handles the orchestration of containers on servers, is by putting code into a container.

Fortunately, there's a lot of help built around packaging code into a container—it is just that essential. So, exactly what is a container? Let's find out. In this article, I'll be introducing containerization, including:

- What containers are
- How they work
- Containers vs VMs
- Enterprise benefits
- Container technologies
- How to get started
- And more

What's a container?

Containers are ways to package code and allow it to run on any machine.

Code can be in different languages, rely on other software, install packages from an internet repo, and require specific system requirements (like memory size) in order to run properly. In containerization, containers hold all of the figurative nuts and bolts needed to run a program. The

container commands its own runtime environment to include things that allow a program to be executable, like:

- Files
- Libraries
- Environment variables

This is different from traditional approaches to virtualization, where each application requires an operating system or an entire virtual machine (VM) to run on the server. Containers are versatile because they can function on:

- Bare-metal servers
- Cloud servers
- A single virtual machine on a server.

So, instead of cooking in another person's kitchen, containers allow coders to create their own kitchens on servers wherever they go. Servers are just personal computers without all the bells and whistles of a sleek design and a fancy OS to make the server user-friendly. In terms of product performance, containerization ensures:

- Reliability
- Consistency

How containers work

Containers will install all the prerequisite files in order to run the software at hand.

Remember buying software in boxes at Fry's Electronics? () On the back of the box, there would usually be specs to the software that said this software will only run on machines that run Mac or Windows. Then, there would be specs about how the software would need a minimum of 512 MB of RAM and 2GB of available hard drive space to run. More complicated apps might require a graphics card with minimum memory.

In a way, the specs on the back of software packages were the way for software developers to limit the risk of the apps failing on the user's devices, especially when they deploy their application widely across many types of computer systems. Today, app developers can:

- Be more precise
- Use remote computers
- Instruct exactly what kind of system their software needs to run on

In fact, with containerization, the developers can build it themselves to ensure the software runs and will not fail.

In containerization, there are two spec files that define what software is required on a machine and what hardware is required to run the machine. They are the:

1. **Dockerfile** which controls the runtime environment and the installation of necessary packages to sufficiently run.
2. **Yaml file** which controls the hardware and the network security requirements

Apart from understanding the framework, you also need to understand the outcomes. After a

container is created, it forms an image. An image results from the complete set of runtime components in a single container. Once the code has been containerized, and the image is created, it can then be deployed on a chosen host. These can be:

- Run locally on each person's machine.
- Deployed on cloud machines.

Containerization vs server virtualization

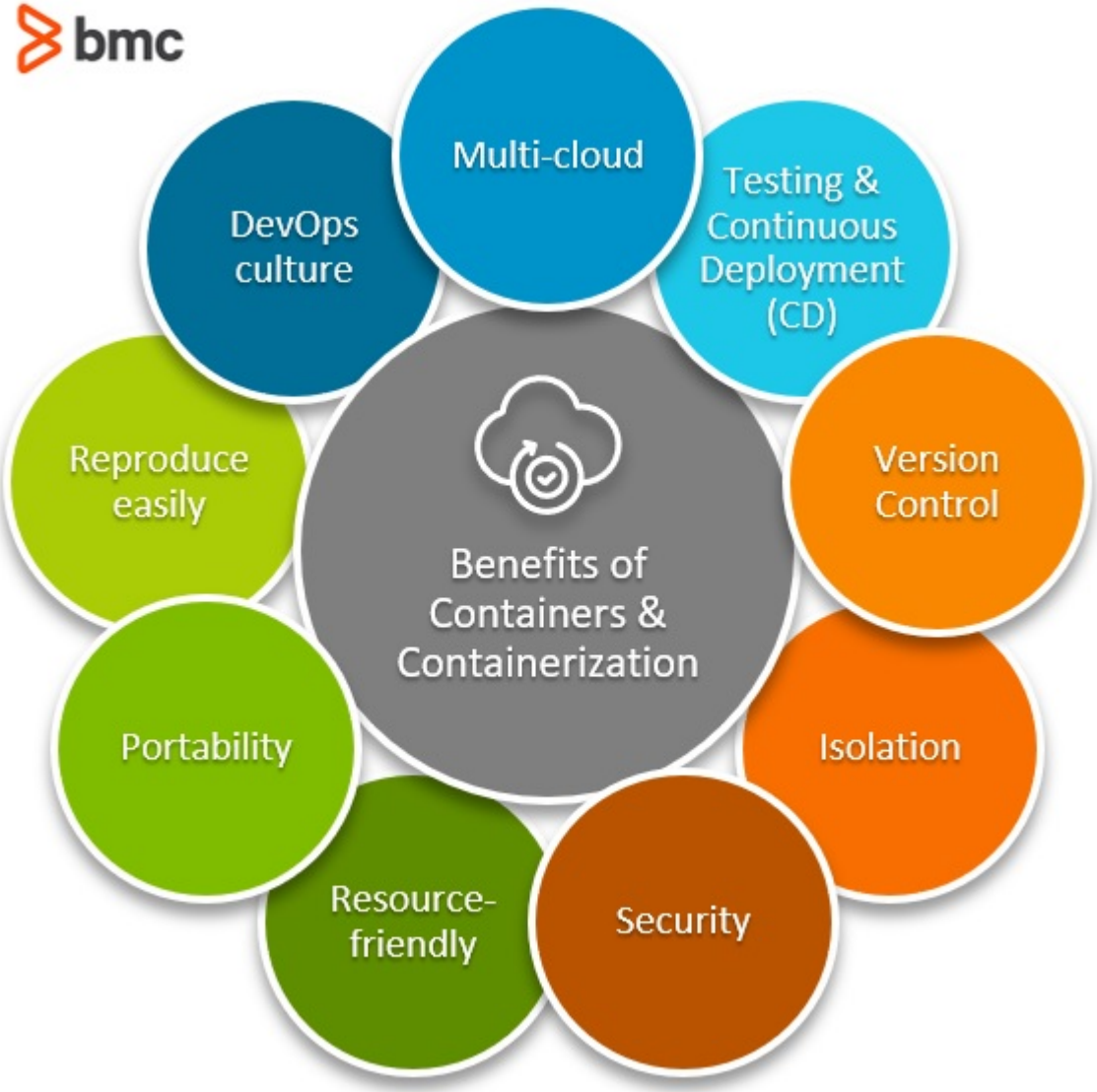
When people talk about server virtualization, they are usually speaking of a virtual machine. A VM creates a hypervisor layer between operation system, applications and services and memory, storage, and the like.

This layer acts as its own VM, creating a self-contained environment to run a single application. Each application that is virtualized consumes its own version of an OS. In order to develop in a virtual environment, you'd need to:

- Have several available versions of operating systems
- Purchase multiple licenses

On the other hand, containers allow multiple applications to run on a single VM. This limits the number of software licenses an enterprise company must invest in to develop in a container environment. They do this by sharing OS kernels, instead of requiring their own. For this reason, developing in a container is a more resourceful approach, both financially and computation, to enterprise software development.

(Understand the differences between containers and VMs.)



Enterprise benefits of containerization

Containers are the answer to a lot of problems when teams are developing code:

- Different code bases
- Sharing code across different systems
- Sharing code in different runtime environments
- Security
- Versioning

Now, let's look at top benefits of deploying software on a container engine.

Multi-cloud platform technology

One of the most desirable benefits of using containerization as a virtualization method is that it can operate on the cloud. Many engines support multi-cloud platforms so they can be run inside platforms like:

- Amazon EC2 instances
- Google Compute Engine instances
- Rackspace servers

- VirtualBox

Containers package code to be shipped by any carrier. It doesn't matter who is handling it, giving the customer more power to pick from different providers.

Testing & Continuous Deployment

Containerization gives enterprise businesses the flexibility to build, test, and release images to deploy on multiple servers.

While consistency across environments sometimes fluctuates, especially when it comes to development and release cycles, container providers like Docker are making it easier to ensure consistency no matter which environment you deploy your image in. That's why containerization is such a good option for organizations using DevOps to accelerate application delivery.

(Explore the DevOps practices of continuous deployment and continuous testing.)

Version control

To remain competitive, container platforms must ensure version control.

Docker set the bar high by offering simplified version control that makes it easy to roll back to a previous image if your environment breaks. Competitors have followed suit, making this method of virtualization good for developers who need version control available at their fingertips.

Isolation

A key component of virtualization is isolation, the act of segregating resources for each application. Container engines perform better than virtual machines when it comes to isolation. But the choice to use one over the other needs consideration, and depends on the use case.

In general, compared to VMs, containers have:

- Faster startup times
- Less redundancy
- Better resource distribution

Security

When Docker debuted, security was thin. Over time, though, Docker has fixed many of the main issues, like running every container from the Root folder.

Providers of containerization offer different ways to ensure security, but one thing is consistent across the board. If one of your containers gets hacked, applications running on other containers are not susceptible.

Still, individual containers are susceptible to attacks, but there exist ways to secure them, through best practices and third-party monitoring services.

(Follow these security best practices for Docker.)

Versatile & resource-friendly

Most developers like containers because they are a versatile, resource-friendly approach to software development. Companies can ensure all of their development and deployment needs are met while conserving server space, whether it's physical, virtual, or cloud.

Compared to VMs, containers:

- Use less space, usually measured in MB rather than GB
- Can be restricted to consume a minimal amount of resources

Portability

Because container applications can run on cloud servers, they are generally more accessible than other applications. Programming in containers offers a portable software development approach.

Reproducibility

DevOps loves containerization, especially because it offers the benefit of reproducibility. Each container's components remain static and unchanging from code to deployment. They create one single image that can be reproduced in other containers over and over again.

Containers support DevOps culture

For any development environment that requires portability and versatility, DevOps and containers offer a fantastic solution. This increasingly widespread adoption is fueled by two key trends:

- Businesses continue to adopt cloud technology widely, allowing you to run operations from the palm of your hands.
- The continued growth in adoption of as-a-service providers that package cloud services to simplify business operations and reduce cost.

While more businesses navigate the tricky waters of making sure their cloud resources are customized, integrated, automated and functioning as they should, the need for qualified DevOps professionals has skyrocketed. With more DevOps professionals usually comes a culture shift, one where two things happen:

- Applications are more and more accessible.
- Companies seek innovative solutions to mitigate rapid growth.

Finally, remote teams and employees increase the demand for portability. They work from remote locations with internet access, often spanning time zones and set working hours, so their contributions can come in at any time).

Containerization offers exactly what they are looking for—a package for software that ship together. Contributions can be made any time. Containers offer consistent performance across time zones and devices.

(Learn more about managing code and containers).

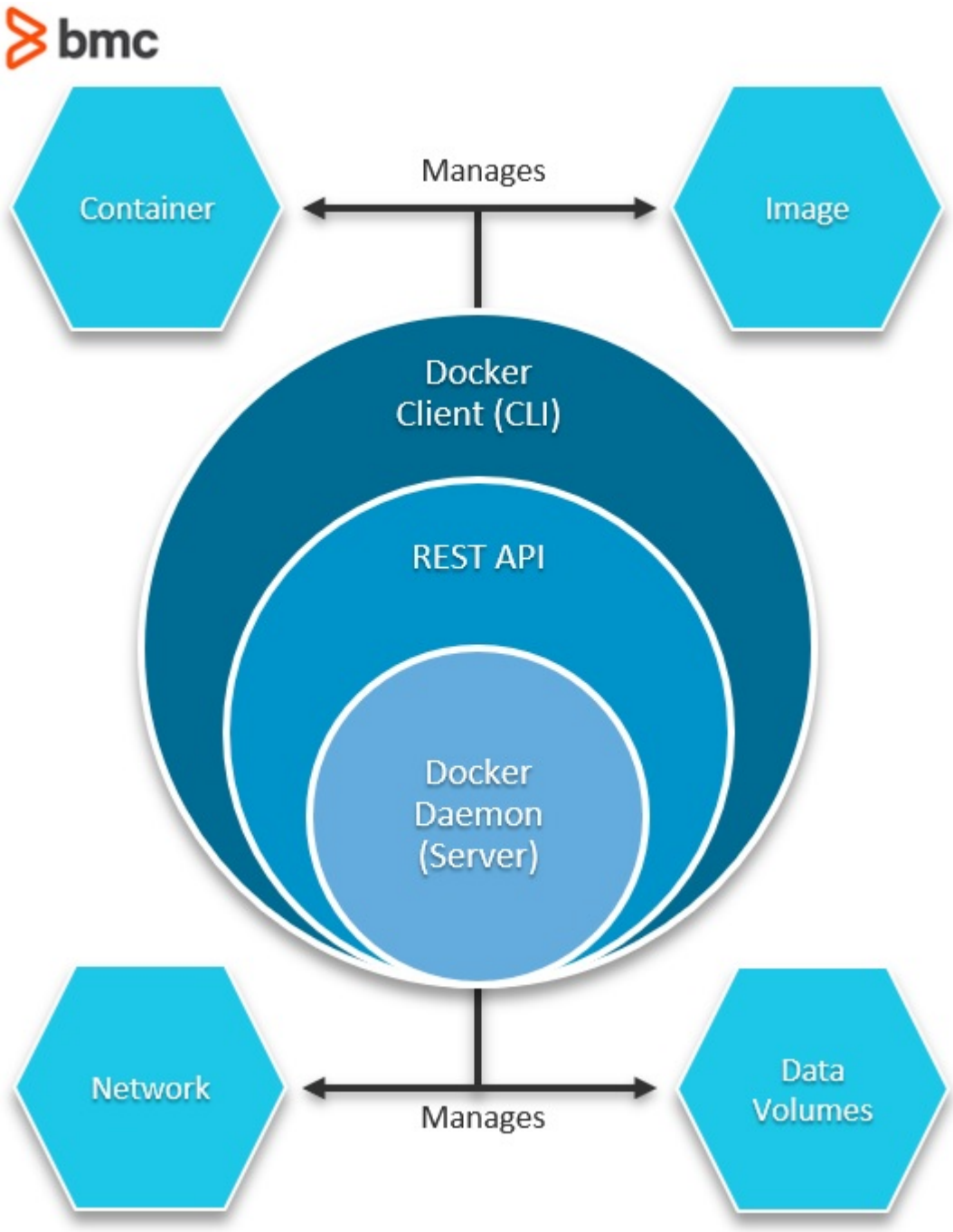
Container systems

A number of services help orchestrate containers, on the cloud in Kubernetes environments or more traditional single server instances. Here's a brief look at a few popular options.

Docker & Kubernetes

The most widely used platform is Docker, an open source container system based on runC. Docker images work on a number of as a service platforms, making them more versatile than some of its competitors.

Docker's basic functionality looks like this:



An emerging best practice is to use Docker and Kubernetes together. Though distinct technologies, integrating K8s and Docker creates an isolation mechanism that lets you augment container resources more efficiently.

CoreOS (rkt)

The biggest competitor to Docker, CoreOS'rkt (pronounced CoreOS "rocket") is a low-level framework that uses systems to create foundational applications. It's designed as the container engine that powers Google Kubernetes.

Google Kubernetes (Cloud Run, GKE & GCE)

Google Kubernetes has its own container engine, rkt, but it is also a community where users can run other popular engines. It's an open source host environment for creating libraries of applications to share or develop.

Kubernetes is a good portable option, since it offers a full cloud server that can be accessed anywhere.

Amazon AWS

AWS offers Backend-as-a-Service (BaaS) that includes a Containers-as-a-Service (CaaS) offering to its customers. It also happens to be a widely used host for those looking to deploy Docker images. Like Kubernetes, AWS offers the portability of cloud computing.

Cloud Foundry

Cloud Foundry's Garden is an open source option that offers containerization as part of its Platform-as-a-Service options (PaaS). Garden is the container engine. Cloud Foundry is the host for developing, testing, and deploying portable applications on a cloud server.

Getting started with containers

Ready to try out containers for yourself? I'll walk you through the process.

Requirements file

Here are commands for how to generate a list of all the installed packages on a device. Presumably, the code has been written in an environment, and compiles fine.

1. Get list of all the installed packages your software depends on—its dependencies.
2. Copy this file to another machine.
3. Install the packages on the machine so it has everything it needs to run.

Below are commands to both get the list of packages and to install the list on another machine. When a container gets launched on a server, it will need to run these commands to install the software on the server it just landed on.

These are examples for Ruby and Python. These steps can be found and repeated for any computer language.

For Python

```
Get all python packages in your environment:
```

```
pip freeze > requirements.txt

Install python requirements on a machine:
pip3 install -r requirements.txt
```

For Ruby

```
Put lists of gems in a file:
xargs gem unpack < $LISTNAME

Build gems without internet connection:
xargs -I gemname gem build gemname/gemname.gemspec < $LISTNAME

Build gems with internet connection:
xargs gem install < $LISTNAME
```

(Follow this excellent how-to on putting Ruby code in a Docker container.)

Docker & Dockerfile code

For Docker, the Dockerfile is a must, but its kind has been repeated for other container registries such as Google's Container Registry.

Important things I learned from using Docker:

1. Use existing Docker images to start, then modify them.
2. Look at the size of the Docker image.
3. It is possible to build one docker image that downloads all the packages, then copy that image into another image to save space. (Sometimes when packages are downloaded and installed, the data is double whenever it's downloaded as a zip and extracted.)

The Dockerfile is a set of instructions to tell the container what to do when it gets built, which happens right before it is deployed.

The Dockerfile uses a combination of FROM, RUN, WORKDIR, COPY, RUN, CMD commands with normal command line inputs afterwards. The following is a simple Dockerfile that:

1. Builds a container from a 12-alpine image.
2. Runs a Python command.
3. Copies everything from the directory to the container under the /app directory.
4. Installs all the requirements.
5. Runs the command node which is meaningful to something in the application itself.

```
'''
 FROM node:12-alpine
 RUN apk add --no-cache python g++ make
 WORKDIR /app
 COPY . .
 RUN pip3 install -r requirements.txt
 CMD
```

```
```

(See Docker docs for more details.)

YAML file

The YAML file is a configuration file that tells the Kubernetes servers exactly what the container's requirements are to run. If you've worked with K8s, you'll already be familiar with YAML.

The YAML file specifies whether the container needs:

- 256 MB or 8 GB of memory
- An attached volume for data
- How many processes it can run
- The security measurements
- And, really, so much more

Short for either or 'yet another markup language' or 'YAML ain't markup language', YAML is a serialization language that is highly readable. It's a bit like JSON, to just store data. While it can be used for many things, it is commonly used for configuration files, such as the config file when containerizing code.

Nick Chase with Marantis breaks down the YAML further. Here, I have only copied what a YAML ends up looking like.

```
```
```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rss-site
  labels:
    app: web
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: front-end
          image: nginx
          ports:
            - containerPort: 80
        - name: rss-reader
          image: nickchase/rss-php-nginx:v1
```

```
          ports:
            - containerPort: 88
'''
```

When to use containers

Containers are perfect for any enterprise or organization that is looking to enhance digital enterprise management via solutions that offer reliability, portability, versatility and reproducibility in a virtual environment.

Companies moving towards a DevOps culture should introduce containers carefully. The following steps are recommended when introducing containers:

1. Evaluate business needs. Practice running containers on a small scale and see how it fits into the business model and culture.
2. Pilot containers with your DevOps team.
3. Move into the Production phase and deploy containers into your infrastructure.

To jump aboard the Kubernetes infrastructure requires two basic skills

- Containerizing your code.
- Learning Kubernetes (Gonna have to learn that kubectl—luckily, our K8s Guide has you covered)

Kubernetes orchestrates the deployment of containers on servers—though there are other options, too. If there are no containers, there is nothing to deploy.

The reason cloud services companies exist is simply because managing servers is costly from a management perspective and costly as an equipment perspective. While they handle most of the cloud problems, it is still up to you to make your containers and deploy them onto their service.

Related reading

- BMC DevOps Blog
- How To Run MongoDB as a Docker Container
- The State of Containers Today: A Report Summary
- How To Introduce Docker Containers in The Enterprise, part of our Docker Guide
- Containers Aren't Always the Solution