INTRODUCTION TO DOCKER: A BEGINNER'S GUIDE

Docker is one of the most popular tools for application containerization. Docker enables efficiency and reduces operational overheads so that any developer, in any dev environment, can build stable and reliable applications.

Let's take a look at Docker, starting with application development *before* Docker.

(*This is part of our Docker Guide. Use the right-hand menu to navigate.)*

App development today

One common challenge for DevOps teams is managing an application's dependencies and technology stack across various cloud and development environments. As part of their routine tasks, they must keep the application operational and stable—regardless of the underlying platform that it runs on.

Development teams, on the other hand, focus on releasing new features and updates. Unfortunately, these often compromise the application's stability by deploying codes that introduce environment-dependent bugs.

To avoid this inefficiency, organizations are increasingly adopting a containerized framework that allows designing a stable framework *without* adding:

- Complexities
- Security vulnerabilities
- Operational loose ends

Put simply, containerization is the process of packaging an application's code—with dependencies, libraries, and configuration files that the application needs to launch and operate efficiently—into a standalone executable unit.

Initially, containers didn't gain much prominence, mostly due to usability issues. However, since Docker entered the scene by addressing these challenges, containers have become practically mainstream.

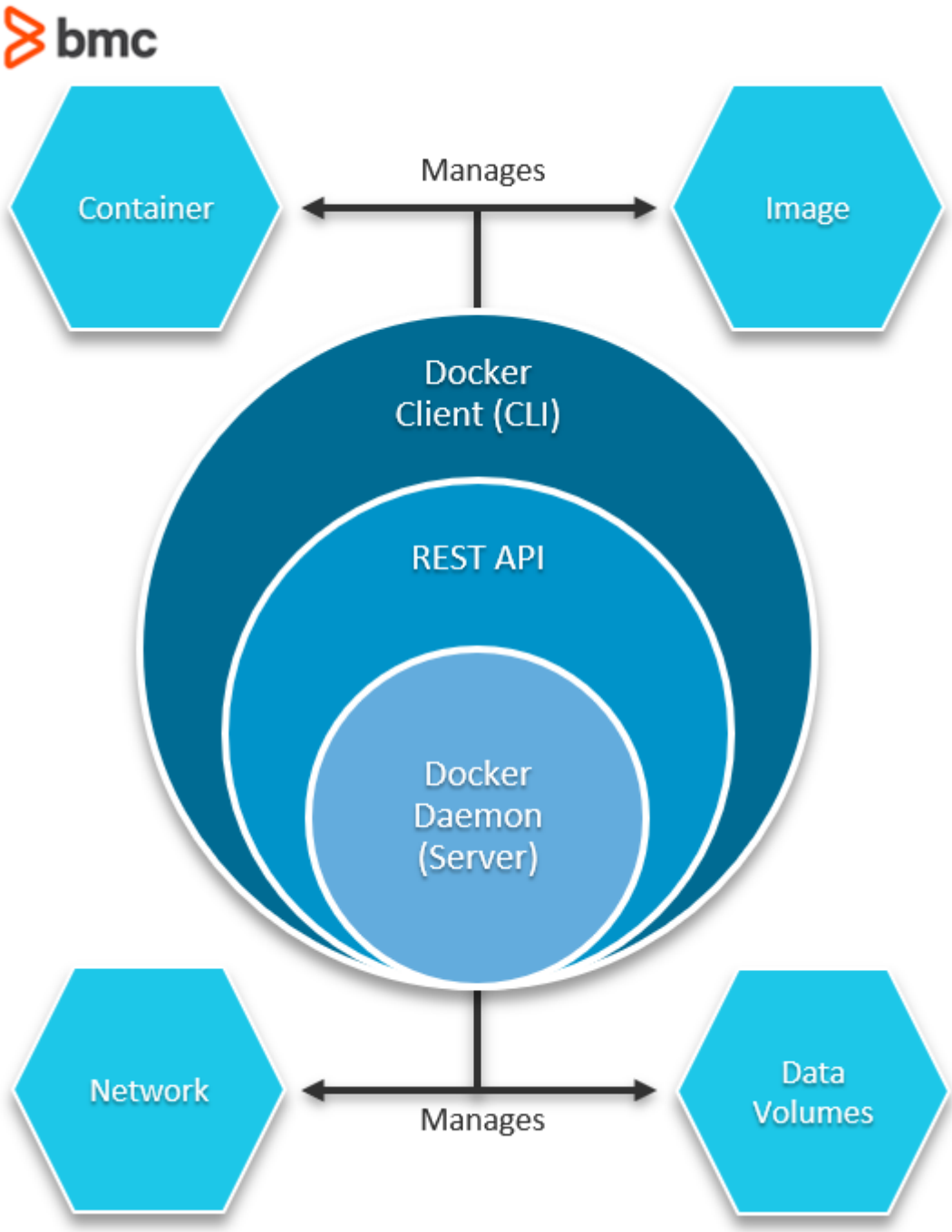
What is Docker?

Docker is a Linux-based, open-source containerization platform that developers use to build, run, and package applications for deployment using containers. Unlike virtual machines, Docker containers offer:

- OS-level abstraction with optimum resource utilization
- Interoperability
- Efficient build and test
- Faster application execution

Fundamentally, Docker containers modularize an application's functionality into multiple components that allow deploying, testing, or scaling them independently when needed.

Take, for instance, a Docker containerized database of an application. With such a framework, you can scale or maintain the database independently from other modules/components of the application *without* impacting the workloads of other critical systems.



Components of a Docker architecture

Docker comprises the following different components within its core architecture:

- Images
- Containers
- Registries
- Docker Engine

Images

Images are like blueprints containing instructions for creating a Docker container. Images define:

- Application dependencies
- The processes that should run when the application launches

You can get images from DockerHub or create your own images by including specific instructions within a file called Dockerfile.

Containers

Containers are live instances of images on which an application or its independent modules are run.

In an object-oriented programming analogy, an image is a class and the container is an instance of that class. This allows operational efficiency by allowing to you to multiple containers from a single image.

Registries

A Docker registry is like a repository of images.

The default registry is the Docker Hub**,** a public registry that stores public and official images for different languages and platforms. By default, a request for an image from Docker is searched within the Docker Hub registry.

You can also own a private registry and configure it to be the default source of images for your custom requirements.

Docker Engine

The Docker Engine is of the core components of a Docker architecture on which the application runs. You could also consider the Docker Engine as the application that's installed on the system that manages containers, images, and builds.

A Docker Engine uses a client-server architecture and consists of the following sub-components:

- **The Docker Daemon** is basically the server that runs on the host machine. It is responsible for building and managing Docker images.
- **The Docker Client** is a command-line interface (CLI) for sending instructions to the Docker Daemon using special Docker commands. Though a client can run on the host machine, it relies on Docker Engine's REST API to connect remotely with the daemon.
- **A REST API** supports interactions between the client and the daemon.

Benefits of Docker in the SDLC

There are numerous benefits that Docker enables across an application architecture. These are some of the benefits that Docker brings across multiple stages of the software development lifecycle (SDLC):

- **Build**. Docker allows development teams to save time, effort, and money by *dockerizing* their applications into single or multiple modules. By taking the initial effort to create an image tailored for an application, a build cycle can avoid the recurring challenge of having multiple versions of dependencies that may cause problems in production.
- **Testing.** With Docker, you can independently test each containerized application (or its components) without impacting other components of the application. This also enables a secured framework by omitting tightly coupled dependencies and enabling superior fault

tolerance.
- **Deploy & maintain.** Docker helps reduce the friction between teams by ensuring consistent versions of libraries and packages are used at every stage of the development process. Besides, deploying an already tested container eliminates the introduction of bugs into the build process, thereby enabling an efficient migration to production.

When it comes to the enterprise use of containers, you can rest easy knowing that Docker works with so many popular tools, including:

- Kubernetes
- Bitbucket
- MongoDB
- VMWare Tanzu
- Redis
- Nginx
- And more!

Docker alternatives

Although Docker is one of the most popular choices for application containerization, there are alternatives:

- **Containerd.** Originally a tool that was part of the Docker ecosystem, this Docker alternative has morphed into its own high-level container runtime. Unlike Docker, which handles network plugins and overlays, Containerd abstracts these functionalities and focuses on running and managing images.
- **LXC/LXD Linux Containers.** An open-source containerization platform with a set of language bindings, libraries, and tools that enables the creation and management of virtual environments. Being tightly bound to the Linux ecosystem, its adoption rate is comparatively limited.
- **Core OS rkt**. Pronounced as "rocket", this is another open-source software containerization alternative to Docker. An essential feature of rkt is that it is arguably a more secure containerization platform that fixes some of the vulnerable flaws within Docker's design.

A few other lesser-known alternatives include OpenVz and RunC.

Docker supports business agility

The idea of an agile, consistent, and independent environment that allowed faster builds and application interoperability turned out to be more challenging in virtual machines than initially thought.

Thanks to Docker, an organization can now fill the gaps left by virtual machines—without duplicating computing resources and while avoiding effort redundancy. In today's cloud native environment, Dockers are synonymous with application efficiency and maintainability.

No wonder organizations continue adopting Docker!

Related reading

- BMC DevOps Blog
- Managing Containers & Code for DevOps
- Docker Commands: A Cheat Sheet
- Docker Security: 14 Best Practices for Securing Docker Containers
- Kubernetes vs Docker Swarm: Comparing Container Orchestration Tools
- State of Containers: A Report Summary