Introducing container technology and Docker

In this document the Container technology and Docker will be introduced.

1. Problems in application development

There are often many roadblocks that stand in the way of easily moving your application through the development cycle and eventually into production. Besides the actual work of developing your application to respond appropriately in each environment, you may also face issues with tracking down dependencies, scaling your application, and updating individual components without affecting the entire application.

Docker containerization and service-oriented design attempts to solve many of these problems. Applications can be broken up into manageable, functional components, packaged individually with all of their dependencies, and deployed on irregular architecture easily. Scaling and updating components is also simplified.

In this guide, we will discuss the benefits of containerization and how Docker helps to solve many of the issues we mentioned above. Docker is the core component in distributed container deployments that provide easy scalability and management.

2. A Brief History of Linux Containerization

Containerization and isolation are not new concepts in the computing world. Some Unix-like operating systems have leveraged mature containerization technologies for over a decade.

In Linux, LXC, the building block that formed the foundation for later containerization technologies was added to the kernel in 2008. LXC combined the use of kernel cgroups (allows for isolating and tracking resource utilization) and namespaces (allows groups to be separated so they cannot "see" each other) to implement lightweight process isolation.

Later, Docker was introduced as a way of simplifying the tooling required to create and manage containers. It initially used ***LXC*** as its default execution driver (it has since developed a library called ***libcontainer*** for this purpose). Docker, while not introducing many new ideas, made them accessible to

the average developer and system administrator by simplifying the process and standardizing on an interface. It spurred a renewed interest in containerization in the Linux world among developers.

3. What Containerization Brings to the Picture

Containers come with many very attractive benefits for both developers and system administrators / operations teams.

Some of the most benefits are listed below.

I. Abstraction of the host system away from the containerized application

Containers are meant to be completely standardized. This means that the container connects to the host and to anything outside of the container using defined interfaces. A containerized application should not rely on or be concerned with details about the underlying host's resources or architecture. This simplifies development assumptions about the operating environment. Likewise, to the host, every container is a black box. It does not care about the details of the application inside.

II. Easy Scalability

One of the benefits of the abstraction between the host system and the containers is that, given the correct application design, scaling can be simple and straight-forward. Service-oriented design (discussed later) combined with containerized applications provide the groundwork for easy scalability.

A developer may run a few containers on their workstation, while this system may be scaled horizontally in a staging or testing area. When the containers go into production, they can scale out again.

III. Simple Dependency Management and Application Versioning

Containers allow a developer to bundle an application or an application component along with all of its dependencies as a unit. The host system does not have to be concerned with the dependencies needed to run a specific application. As long as it can run Docker, it should be able to run all Docker containers.

This makes dependency management easy and also simplifies application version management as well. Host systems and operations teams are no longer responsible for managing the dependency needs of an application because, apart from a reliance on related containers, they should all be contained within the container itself.

IV. Extremely lightweight, isolated execution environments

While containers do not provide the same level of isolation and resource management as virtualization technologies, what they win from the trade off is an extremely lightweight execution environment. Containers are isolated at the process level, sharing the host's kernel. This means that the container itself does not include a complete operating system, leading to almost instant startup times. Developers can easily run hundreds of containers from their workstation without an issue.

V. *Shared Layering*

Containers are lightweight in a different sense in that they are committed in "layers". If multiple containers are based on the same layer, they can share the underlying layer without duplication, leading to very minimal disk space utilization for later images.

VI. *Composability and Predictability*

Docker files allow users to define the exact actions needed to create a new container image. This allows you to write your execution environment as if it were code, storing it in version control if desirable. The same Docker file built in the same environment will always produce an identical container image.

VII. *Using Dockerfiles for Repeatable, Consistent Builds*

While it is possible to create container images using an interactive process, it is often better to place the configuration steps within a Dockerfile once the necessary steps are known. Dockerfiles are simple build files that describe how to create a container image from a known starting point.

Dockerfiles are incredible useful and fairly easy to master. Some of the benefits they provide are:

- **Easy versioning**: The Dockerfiles themselves can be committed to version control to track changes and revert any mistakes
- **Predicatability**: Building images from a Dockerfile helps remove human error from the image creation process.
- **Accountability**: If you plan on sharing your images, it is often a good idea to provide the Dockerfile that created the image as a way for other users to audit the process. It basically provides a command history of the steps taken to create the image.
- **Flexibility**: Creating images from a Dockerfile allows you to override the defaults that interactive builds are given. This means that you do not have to provide as many runtime options to get the image to function as intended.

Dockerfiles are a great tool for automating container image building to establish a repeatable process.

4. *The Architecture of Containerized Applications*

When designing applications to be deployed within containers, one of the first areas of concern is the actual architecture of the application. Generally, containerized applications work best when implementing a service-oriented design.

Service-oriented applications break the functionality of a system into discrete components that communicate with each other over well-defined interfaces. Container technology itself encourages this type of design because it allows each component to scale out or upgrade independently.

Applications implementing this type of design should have the following qualities:

- They should not care about or rely on any specifics of the host system
- Each component should provide consistent APIs that consumers can use to access the service
- Each service should take cues from environmental variables during initial configuration
- Application data should be stored outside of the container on mounted volumes or in data containers

These strategies allow each component to be independently swapped out or upgraded as long as the API is maintained. They also lend themselves towards focused horizontal scalability due to the fact that each component can be scaled according to the bottleneck being experienced.

Rather than hard coding specific values, each component generally can define reasonable defaults. The component can use these as fallback values, but should prefer values that it can gather from its environment. This is often accomplished through the aid of service discovery tools, which the component can query during its startup procedure.

Taking the configuration out of the actual container and placing it into the environment allows for easy changes to application behavior without rebuilding the container image. It also allows a single setting to influence multiple instances of a component. In general, service-oriented design couples well with environmental configuration strategies because both allow for more flexible deployments and more straight-forward scaling.

5. *The Docker platform*

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight because they don’t need the extra load of a hypervisor, but run directly

within the host machine's kernel. This means you can run more containers on a given hardware combination than if you were using virtual machines. You can even run Docker containers within host machines that are actually virtual machines!

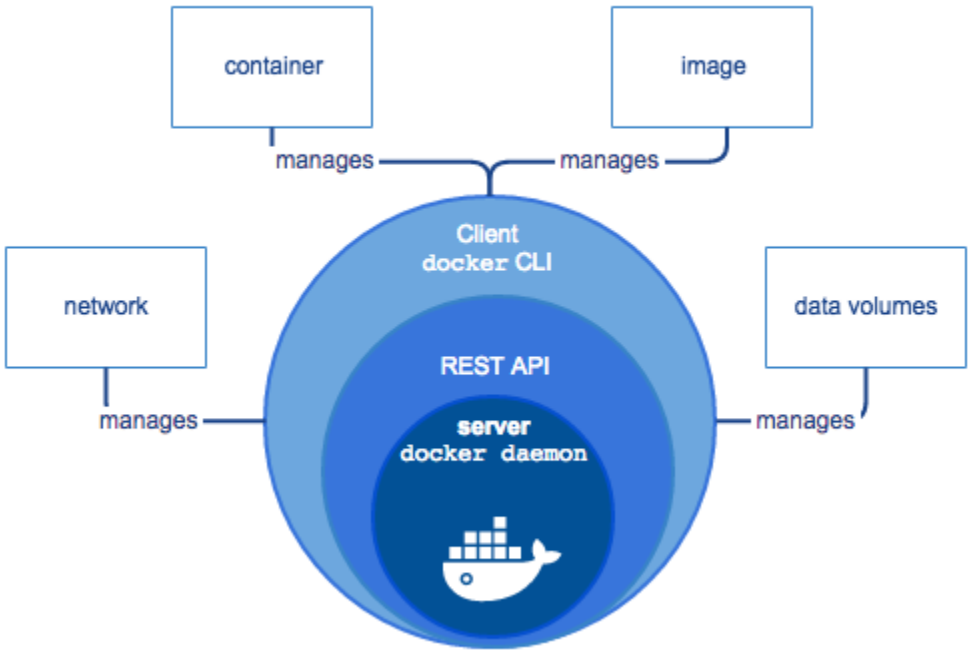
Docker provides tooling and a platform to manage the lifecycle of your containers:

- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

A. *Docker Engine*

Docker Engine is a client-server application with these major components:

- A server which is a type of long-running program called a daemon process (the dockerd command).
- A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface (CLI) client (the docker command).



The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands. Many other Docker applications use the underlying API and CLI.

The daemon creates and manages Docker *objects*, such as images, containers, networks, and volumes.

Note: Docker is licensed under the open source Apache 2.0 license.

B. What can I use Docker for?

Fast, consistent delivery of your applications

Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide your applications and services. Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

Consider the following example scenario:

- Your developers write code locally and share their work with their colleagues using Docker containers.
- They use Docker to push their applications into a test environment and execute automated and manual tests.
- When developers find bugs, they can fix them in the development environment and redeploy them to the test environment for testing and validation.
- When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.

Responsive deployment and scaling

Docker's container-based platform allows for highly portable workloads. Docker containers can run on a developer's local laptop, on physical or virtual machines in a data center, on cloud providers, or in a mixture of environments.

Docker's portability and lightweight nature also make it easy to dynamically manage workloads, scaling up or tearing down applications and services as business needs dictate, in near real time.

Running more workloads on the same hardware

Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines, so you can use more of your compute capacity to achieve your business goals. Docker is perfect for high density environments and for small and medium deployments where you need to do more with fewer resources.

6. Docker architecture

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

A. The Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

B. The Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

C. Docker registries

A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry. If you use Docker Datacenter (DDC), it includes Docker Trusted Registry (DTR).

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

D. Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

- **IMAGES**

An *image* is a read-only template with instructions for creating a Docker container. Often, an image is *based on* another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a *Dockerfile* with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

- **CONTAINERS**

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear

7. Conclusion

Docker provides the fundamental building block necessary for distributed container deployments. By packaging application components in their own containers, horizontal scaling becomes a simple process of spinning up or shutting down multiple instances of each component. Docker provides the tools necessary to not only build containers, but also manage and share them with new users or hosts.

8. References

https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-overview-of-containerization

https://docs.docker.com/engine/docker-overview/