Docker: Packing Application and Dependency to avoid platform dependency.

Example of platform Dependency: OS restart, downloading external packages.

Comparison points in monolithic vs microservice: Size, startup and integration.

Stangler pattern: 1 by 1 to avoid downtime.

Manual Deployment vs container Deployment

**What Is Containerd and runc?**

When Docker was monolithic, a single application translated our command, then pulled in the container image, started it, and made it accessible on port 80. Nowadays, that's not true anymore. In a very simplified form, this is what currently happens:

The **Docker CLI** utility accepts the command. Then it figures out what we want to do. After it understands our intention, it passes this intention to the **Docker Daemon**. This daemon is a separate program (from Docker CLI) that always runs in the background, waiting for instructions. After the Docker Daemon receives our desired action, it tells another app, called a **container runtime**, to pull in the container image. This container runtime is called **containerd**.

So we can now finally understand what contained is. In tech terms, it is a container runtime. This is a sort of container manager. It takes care of things such as:

* Downloading container images.
* Uploading container images.
* Setting up networking between these containers, so that they can communicate with each other, or the outside world.
* Managing data and files stored inside these containers.
* Starting, stopping, restarting containers.

containerd is called a high-level container runtime. For some actions, it makes use of yet another runtime, called a low-level container runtime. This low-level runtime is called **runc**. For example, when containerd needs to start a container, it tells runc to do that.

libcontainer: libcontainer refers to a container runtime library that was originally developed by Docker. It provides a set of tools for working with Linux containers, offering an interface for container-related operations such as container creation, starting, stopping, and resource isolation. libcontainer was designed to be lightweight and flexible, enabling the implementation of container technologies.

Oci: OCI, or the Open Container Initiative, is an open standard for container formats and runtimes. The initiative was launched to establish a set of common, minimal standards for container technology, fostering compatibility and interoperability across different container platforms. OCI is hosted by the Linux Foundation.

docker shim: The "shim" in the context of Docker refers to a small piece of software that serves as a bridge between Docker and container runtimes. The purpose of the Docker shim is to maintain compatibility with the Open Container Initiative (OCI) runtime specification. The OCI specification defines standards for container runtimes, and Docker aims to adhere to these standards.

Appc: appc, short for "App Container," is a specification for container images and the runtime environment of containers. It was originally developed by CoreOS as an open standard for defining and running application containers. The App Container (appc) specification aims to provide a common and minimal standard for container technologies, promoting interoperability and compatibility across different container runtimes.

Grpc: gRPC is an open-source remote procedure call (RPC) framework developed by Google. It is part of the Cloud Native Computing Foundation (CNCF) and is designed to enable efficient and scalable communication between distributed systems. gRPC is often used in microservices architectures and other scenarios where services need to communicate with each other over a network.

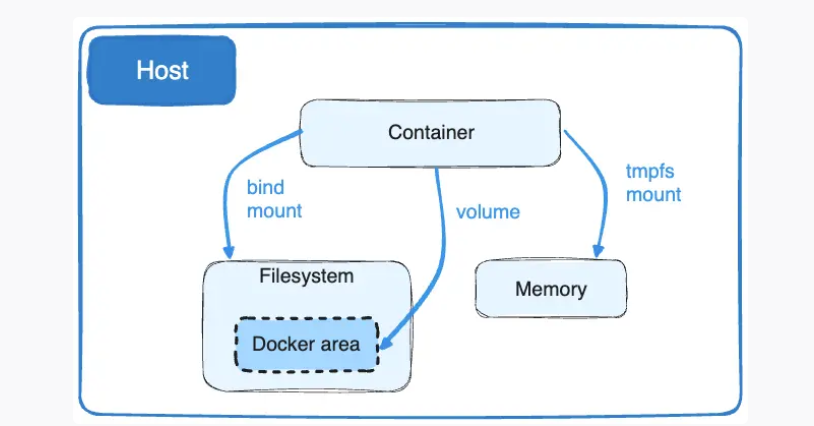
rkt containers

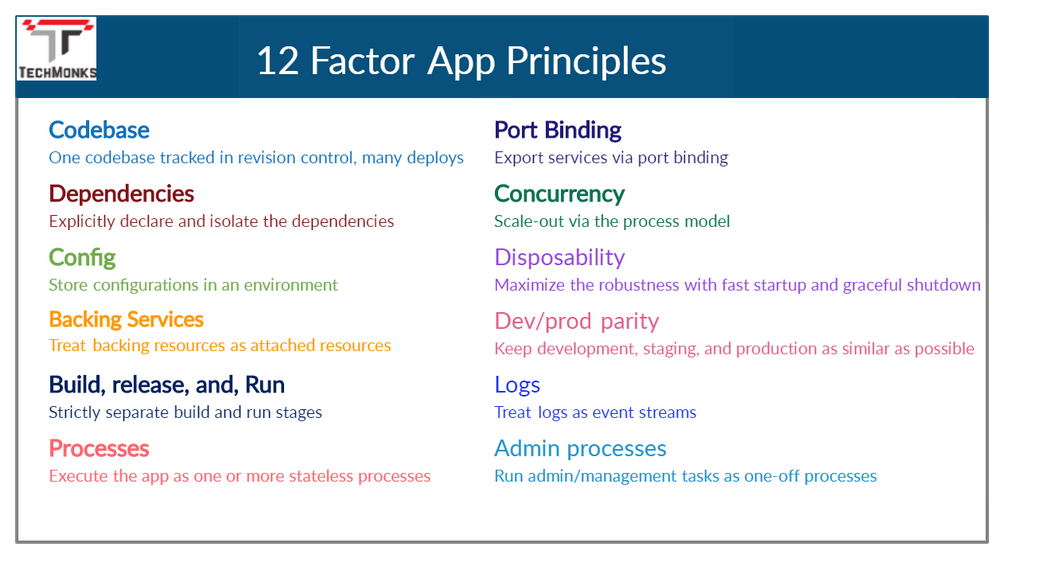
rkt (pronounced "rocket") was a container runtime developed by CoreOS. rkt was designed with a focus on simplicity, security, and composability. It aimed to provide an alternative to Docker, with a particular emphasis on security features. Key features of rkt included:

1. **Security:** rkt implemented security measures such as running each container in its own process tree and using the App Container (appc) specification to define container images.
2. **Composability:** rkt was designed to be composable, allowing users to select different components (like image discovery, fetching, and running) independently, unlike Docker, which was more monolithic.
3. **Integration with systemd:** rkt could be easily integrated with systemd for process management and control.
4. **Stage1 and Stage2 concepts:** rkt introduced the concept of "stage1" and "stage2" to separate the responsibility of fetching and running containers. The stage1 component was responsible for downloading and validating the container image, while stage2 was responsible for executing the container.

Docker lifecycle states:

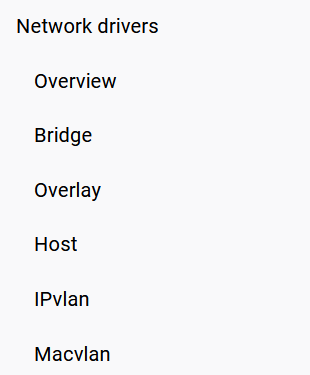
* 1. Creating
  2. Created
  3. Running
  4. Paused
  5. Stopped
  6. Deleted





Opting Docker vs VM:

The primary factor is what is important for you application or infrastructure. if application go for Docker else continue to container.

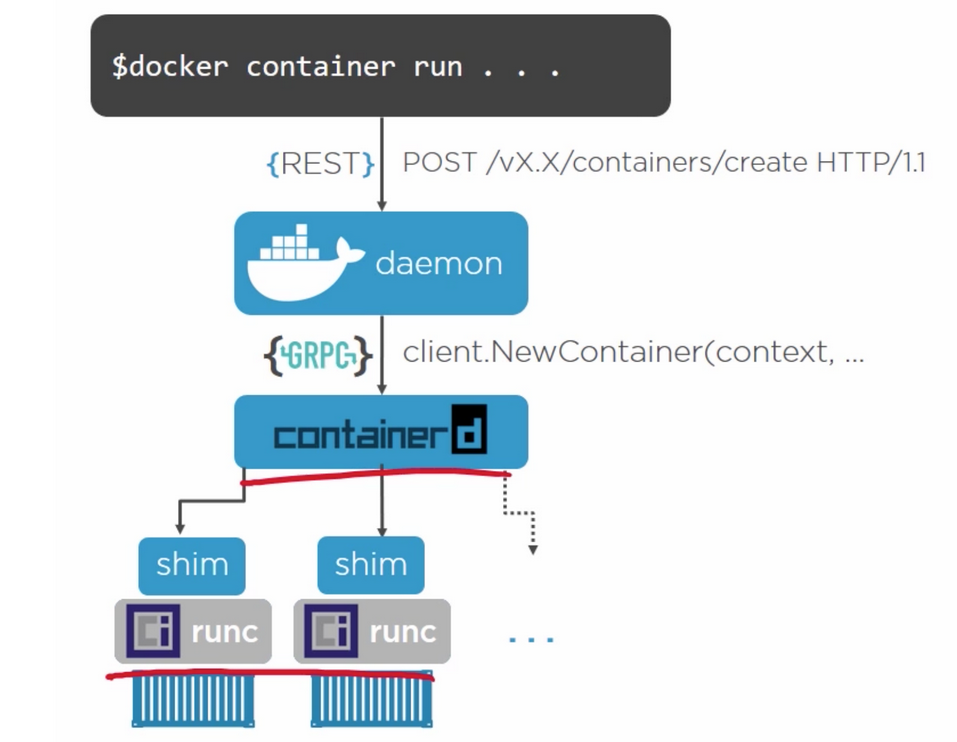


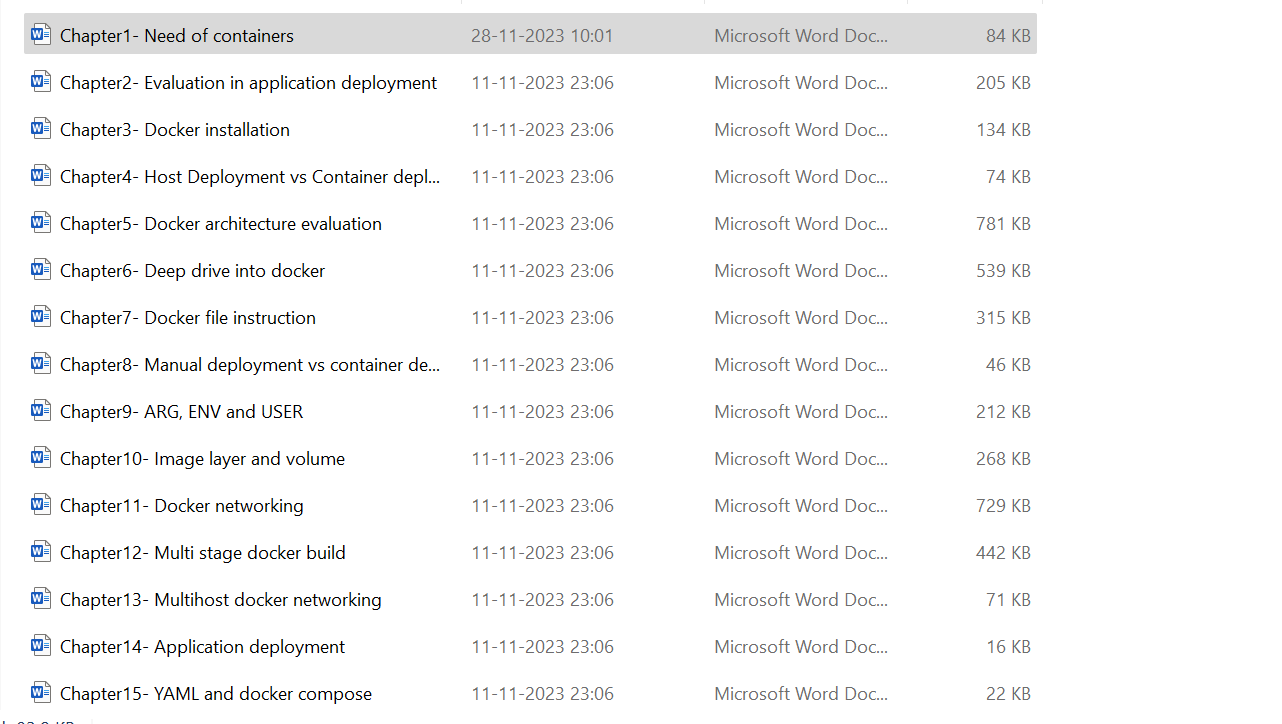
Some applications, especially legacy applications or applications which monitor network traffic, expect to be directly connected to the physical network. In this type of situation, you can use the macvlan network driver

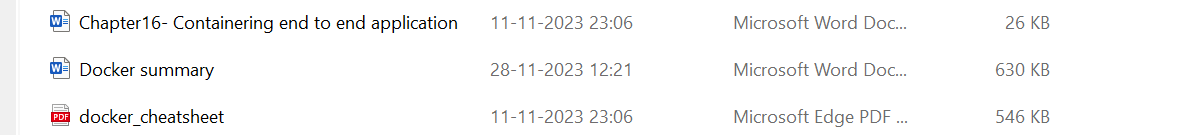
Operation performed by runc:

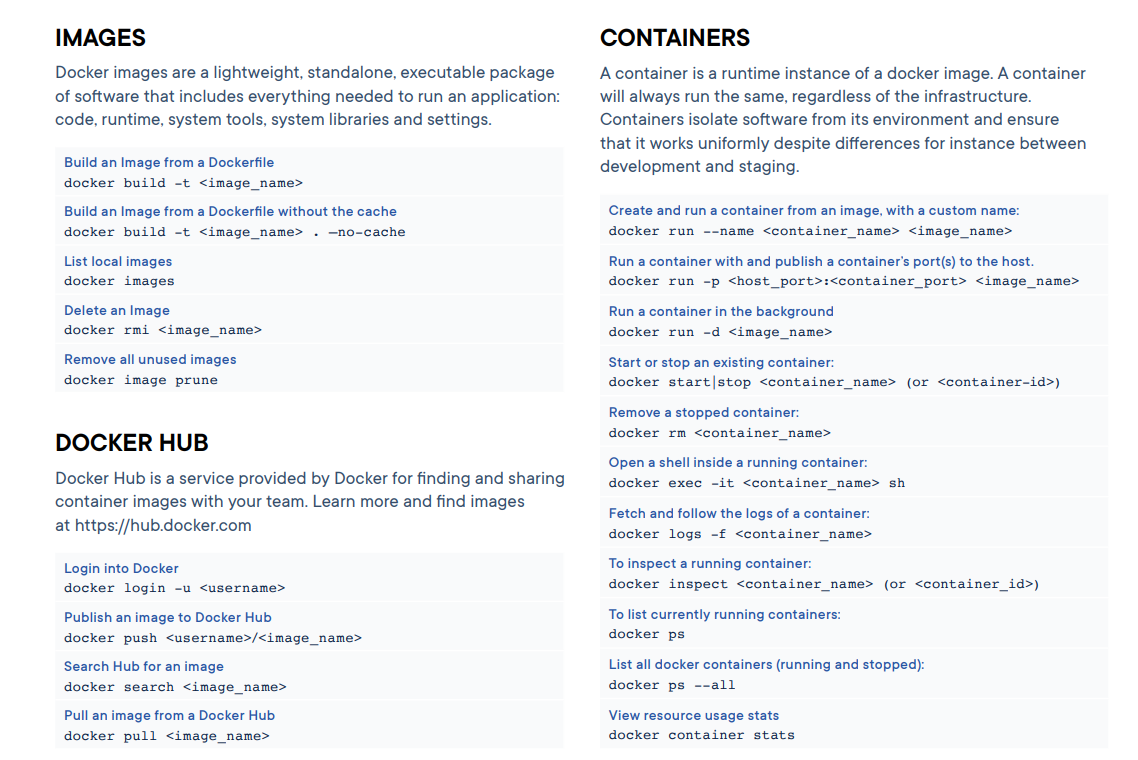
Here are some of the key operations performed by runc:

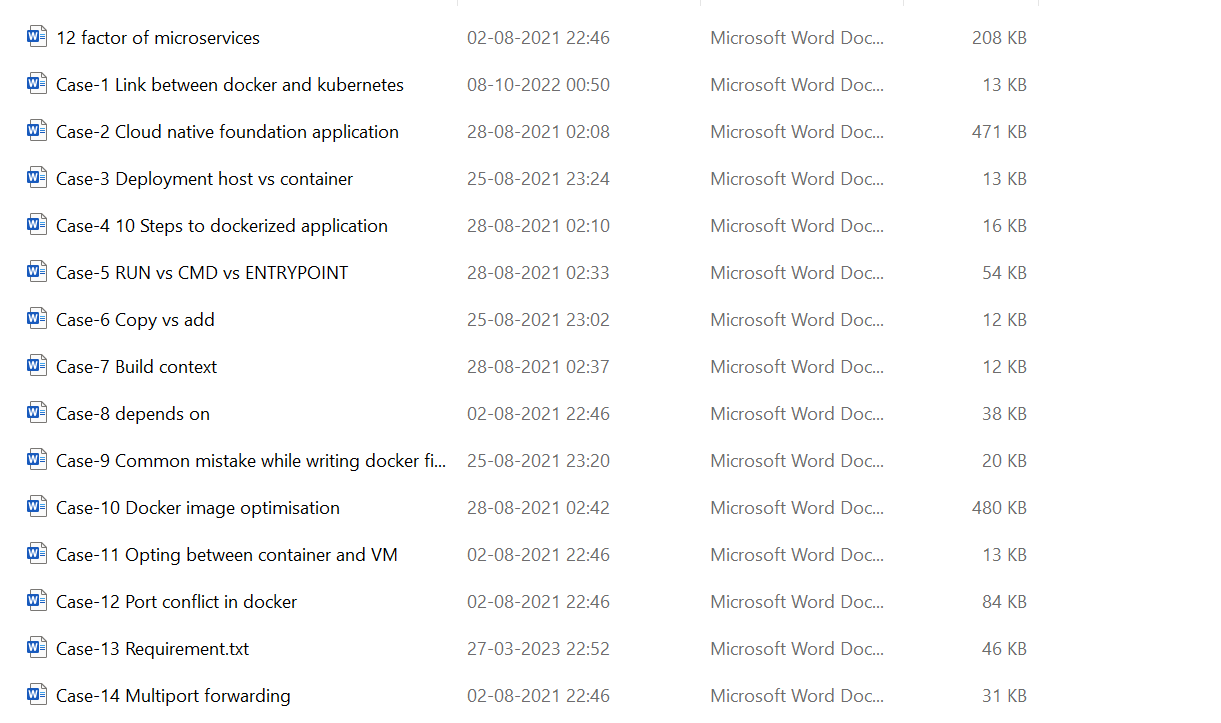
1. **Container Lifecycle Management:**
   * **Create:** runc creates a new container by setting up the container's root filesystem and configuration.
   * **Start:** Initiates the execution of the container's main process.
   * **Exec:** Runs a new process inside an already running container.
2. **Namespace and Cgroups Isolation:**
   * runc utilizes Linux namespaces to provide process, network, and mount isolation. This ensures that processes inside a container have their own view of the system.
   * It uses cgroups (control groups) to manage resource constraints and accounting for the container processes.
3. **Filesystem Setup:**
   * runc sets up the container's root filesystem using a bundle of files. This bundle typically includes the container's root filesystem, a configuration file, and other necessary files.
4. **Process Execution:**
   * runc is responsible for launching and managing the container processes. It sets up the initial process (usually specified in the container's configuration) and monitors its lifecycle.
5. **Signal Handling:**
   * runc manages the propagation of signals to the processes inside the container, ensuring proper termination and cleanup.
6. **Resource Management:**
   * Utilizing cgroups, runc enforces resource constraints on the container processes, such as CPU, memory, and I/O limits.
7. **Runtime Configuration:**
   * runc reads the container's configuration from a JSON file, which specifies various settings for the container, such as environment variables, working directory, and more.

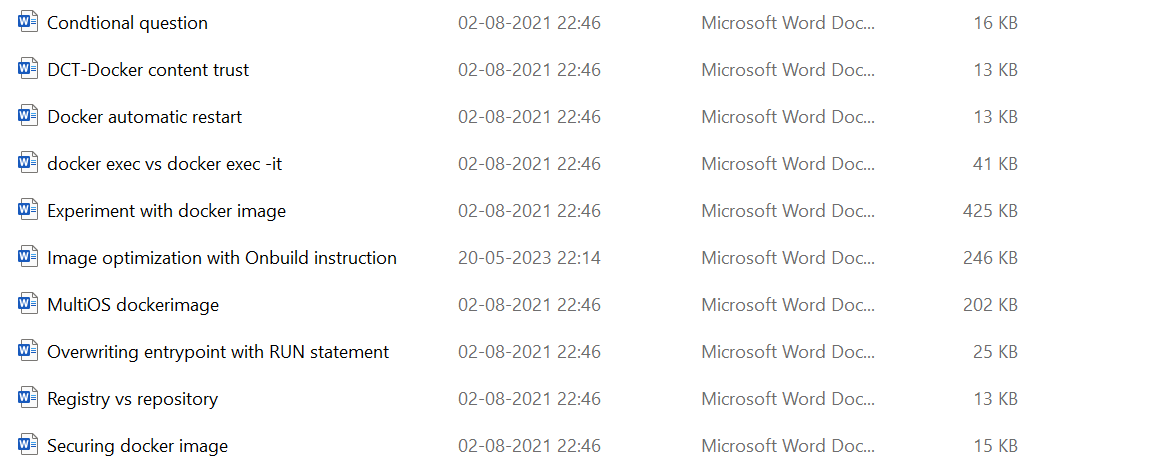












**Container Image Scanning:**

* Use container image scanning tools to identify vulnerabilities and security issues in your container images. Tools like Clair, Trivy, and Anchore can be integrated into your CI/CD pipeline.
* Example Trivy command to scan a Docker image:

bash

trivy image myregistry/myimage:mytag

