Running Docker Containers on IBM Z

October 2018
Running Docker Containers on IBM Z

October 2018
Before using this information and the product it supports, read the information in "Notices" on page 39.
# Contents

**About this document** ........................................... v
Other publications for Linux on Z and LinuxONE ........................................ v

## Chapter 1. Docker basics ........................................ 1
Docker host and Docker engine ........................................ 1
Docker images ........................................ 2
Docker containers ........................................ 2
Registries ........................................ 3

## Chapter 2. Components in a Docker environment ............ 5
Registries ........................................ 6
Clustering and cluster management ........................................ 6
Operations ........................................ 7
Cloud integration ........................................ 7

## Chapter 3. Planning for Docker ................................ 9
Docker containers versus virtual machines ........................................ 9
Obtaining the Docker product ........................................ 10
Getting support ........................................ 10
Host setup ........................................ 11
Scaling with LPAR or virtual machines ........................................ 11
Clustering options ........................................ 12

docker in swarm mode ........................................ 13

Kubernetes ........................................ 13
Page cache sharing ........................................ 14
Storage drivers ........................................ 15
Designing images for sharing ........................................ 15

## Chapter 4. Managing images .................................. 19
Acquiring images ........................................ 19

Using existing images ........................................ 19
Extending existing images ........................................ 19
Base images for the Linux for Z distributions ........................................ 21
Creating an image from scratch ........................................ 22
Multi-arch images ........................................ 22
Keeping images current ........................................ 22

## Chapter 5. Security ........................................ 25
Image security ........................................ 25
Container security ........................................ 25
Docker host ........................................ 27
Mainframe cryptographic features ........................................ 27
CPACF ........................................ 27
Crypto Express adapters ........................................ 28
Software stack ........................................ 29
openCryptoki ........................................ 29
Clear key asymmetric cryptography ........................................ 30
Cryptography with secure and protected keys ........................................ 31
Random data ........................................ 34

## Chapter 6. Avoiding common pitfalls ......................... 35
Use applicable images ........................................ 35
Avoid remote container access ........................................ 35
Use a host kernel with Docker support ........................................ 35

## Accessibility ........................................ 37

## Notices ........................................ 39
Trademarks ........................................ 40

## Index ........................................ 41

© Copyright IBM Corp. 2017, 2018
About this document

This document gives a brief overview of the Docker concepts and summarizes what you need to know to run Docker containers on IBM® mainframes.

Throughout this publication, mainframe is used to include both IBM Z® and IBM LinuxONE™.

Technical changes and new content included in this edition are indicated by a vertical line on the left margin.

You can find the latest version of this document on IBM Knowledge Center at www.ibm.com/support/knowledgecenter/linuxonibm/liaaf/lnz_r_vd.html or on developerWorks® at www.ibm.com/developerworks/linux/linux390/documentation_dev.html.

Other publications for Linux on Z and LinuxONE

You can find publications for Linux on Z and LinuxONE on IBM Knowledge Center and on developerWorks.

These publications are available on IBM Knowledge Center at www.ibm.com/support/knowledgecenter/linuxonibm/liaaf/lnz_r_lib.html:
- Device Drivers, Features, and Commands (distribution-specific editions)
- Using the Dump Tools (distribution-specific editions)
- Running Docker Containers on IBM Z, SC34-2781
- KVM Virtual Server Quick Start, SC34-2753
- KVM Virtual Server Management, SC34-2752
- How to use FC-attached SCSI devices with Linux on z Systems, SC33-8413
- libica Programmer’s Reference, SC34-2602
- Exploiting Enterprise PKCS #11 using openCryptoki, SC34-2713
- Secure Key Solution with the Common Cryptographic Architecture Application Programmer’s Guide, SC33-8294
- Pervasive Encryption for Data Volumes, SC34-2782
- Linux on z Systems Troubleshooting, SC34-2612
- Kernel Messages, SC34-2599
- How to Improve Performance with PAV, SC33-8414
- How to Set up a Terminal Server Environment on z/VM, SC34-2596

You can also find these publications on developerWorks at www.ibm.com/developerworks/linux/linux390/documentation_dev.html.

For versions of documents that have been adapted to a particular distribution, see one of the following web pages:
- www.ibm.com/developerworks/linux/linux390/documentation_ubuntu.html

© Copyright IBM Corp. 2017, 2018
Chapter 1. Docker basics

Containers, an increasingly popular technology for deploying and running software on Linux, can run on IBM LinuxONE and IBM Z.

With containers, you can run multiple encapsulated workloads on a single Linux instance. Docker is a leading software container platform. Docker uses copy-on-write and overlay file system technology, to efficiently deploy and run workloads in containers. For isolation it uses Linux resource scoping mechanisms.

On an IBM mainframe, Linux instances with Docker containers can run in LPAR mode or as a guest of z/VM® or KVM. The Docker CLI and REST API are identical across hardware systems, including the mainframe.

Docker itself does not depend on a particular mainframe level, but it has requirements on Linux. The Linux instance must be compiled to include all modules and functions that are used by Docker. Recent versions of all distributions for IBM mainframes can run Docker.

The sections that follow provide a brief introduction to Docker as applicable to Linux on Z and LinuxONE. For details and a more general view, go to www.docker.com

Docker host and Docker engine

A Docker host is a physical or virtual server on which the core component of Docker runs, the Docker engine. The Docker engine encapsulates and runs workloads in Docker containers.

Like a hypervisor, the Docker engine provides portions of the host's computing resources to its containers, and it isolates containers from one another.

Unlike a hypervisor, the Docker engine does not run separate guest operating systems for its containers. All containers share the same Docker host kernel. As a consequence, workloads in containers cannot use features that are not supported by the Docker host kernel. For the broadest workload potential, use the most recent version of your distribution.
Docker uses Linux features to isolate containers from one another. These features include control groups, namespaces, and pivot_root(). Similar to chroot(), pivot_root() restricts the container view of the root file system to a branch of the host's file system.

---

**Docker images**

*Images serve as read-only templates for the root file systems of Docker containers.*

Images can be of varying complexity. With the overlay capabilities of union file systems, Docker can build complex images as a stack of image layers. An existing image can serve as a *parent image* to which layers are added. The same parent image can be reused in multiple descendants. This capability helps to avoid redundancy and simplifies maintenance.

The bottom layer of an image stack is a *base image*. Base images are built from scratch and, therefore, do not have parent images.

A typical base image for IBM Z or LinuxONE is a basic root file system of a Linux distribution for the mainframe. The base image distribution need not match the distribution of the host Linux kernel. You can use base images for different distributions on the same Docker host.

A complete image stack, typically, amounts to a Linux file system environment with all program files, configuration files, libraries, and utilities that are required to run a specific application. Such stacks capture all dependencies between software components, which makes the captured application self-contained and readily deployable. The image can, for example, contain a Java™ application with the required Java version and all dependent libraries as part of the stack.

The images that you use must be suitable for your mainframe environment. For example, the image must be compiled for the z/Architecture® or it must be a *mult-arch image* that includes support for the z/Architecture.

For more information about images and requirements for images, see Chapter 4, "Managing images," on page 19 and "Use applicable images" on page 35.

---

**Docker containers**

Docker *containers* are created from Docker images.

Other than an image, which is static, a container has a runtime environment and a container-specific read/write layer on top of an image stack. Multiple container instances can be created from the same image. Each container instance then has its own ID and writable layer atop the shared image stack.

The Docker engine allocates resources to a container when the container is started. Starting a container is fast because the writable file system layer uses copy-on-write files. Writable copies of the files are created in the top layer only as they are being modified. After a container is created, it can be started and stopped, and its writable layer persists until the container is deleted.

Docker is designed to support a microservices model according to which a container, ideally, provides just one function. This separation is affordable because a host, in principle, can run numerous Docker containers at an acceptable cost in
terms of CPU and memory. However, the suitable level of fragmentation depends on the workload. The cost of network traffic between containers can be considerable for some workloads.

If you need a customized runtime environment, you can use system containers. Images for system containers are set up to run an initialization process when a container is started. The initialization process can be driven, for example, through systemd.

In system containers, you can run daemons and administration tools without affecting other containers. Administrative tasks can then be performed similar to what you would have in a virtual machine. In terms of behavior, performance, resource consumption, and isolation, system containers usually range between regular containers and virtual machines.

## Registries

A Registry is a hosted service for storing and retrieving images through a registry API.

Public registries are available on the internet. Alternatively, you can maintain private registries at your installation. Typical mainframe environments have security policies that prohibit direct use of public registries. Images can then be added to a private registry after a vetting process through an authorized administrator. You can also create your own images and add them to your private registry.

Each Docker host maintains a local storage area for its image and container layers, usually at /var/lib/docker. This local storage area is populated when images are pulled from registries or when a container is started for the first time.

Do not confuse registries with repositories. In Docker, a repository is a set of Docker images that can be pushed to a registry. The images in a repository have the same name but different tags. Tags are labels for variants of an image. For example, tags can label different code versions, architecture support, or possible extensions of an image.

For examples of registries, see "Registries" on page 6.
Chapter 2. Components in a Docker environment

Typical Docker setups include multiple Docker hosts that are coordinated through a clustering technology and supported through management and infrastructural components.

With the components you include in your Docker environment you can stay close to what is provided and endorsed by Docker Inc., align yourself with what is favored by your distribution, or mix and choose according to your own preferences.

The following figure sketches the major areas of a Docker environment and shows some of the most popular components that are relevant to the mainframe. Kubernetes and swarmkit are alternative clustering technologies. UCP can be used only with swarmkit and Kubernetes has its own dashboard. IBM Cloud Private builds on Kubernetes. The other components can be used together with both clustering technologies.

Because the Docker ecosystem is rapidly evolving, the figure does not claim completeness or currency.
The Docker engine is the core component that runs on every Docker host within a Docker setup. It builds on the containerd daemon and runc to instantiate and run containers. It usually uses one or more registries as a source of images, and it offers a CLI and an API to management components. The Docker engine also includes swarmkit, a clustering technology that you can enable. Alternatives to the Docker engine, like LXC/LXD, rkt, or systemd-nspawn are beyond the scope of this publication.

Registries

Public registries are available on the internet, but you can also use private registries at your installation.

Public registries

Docker Hub and Docker Store

Docker maintains two public registries: Docker Hub and Docker Store. Docker Hub is a community site to which everybody can contribute. Docker Store is more closely controlled by Docker Inc. Many of the images on Docker Store are contributed and signed by reputed companies and some of the images are certified by Docker Inc.

Both registries can be accessed through store.docker.com.

Private registries

Docker Trusted Registry (DTR)

With Docker Enterprise Edition for Linux on z Systems® and LinuxONE, you can purchase DTR, a private registry that includes the Notary tool, which handles digital signatures for images. DTR can run in a Docker container on the mainframe.

For more information, see docs.docker.com/datacenter/dtr/2.2/guides and docs.docker.com/notary/getting_started.

Open Source Registry

You can build your own private registry from the Docker Distribution project on GitHub at github.com/docker/distribution. The Open Source Registry can run in a Docker container on the mainframe.

Clustering and cluster management

Your cluster management tool depends on your clustering technology.

Universal Control Plane (UCP)

With UCP, you set up a swarm, a cluster of Docker hosts that is based on swarmkit, Docker’s clustering technology.

You can obtain UCP with Docker Enterprise Edition for Linux on z Systems and LinuxONE. UCP can run in a Docker container on the mainframe.

For more information about UCP, see docs.docker.com/datacenter/ucp/2.0/guides.

Kubernetes and Dashboard

Kubernetes is a community-maintained alternative clustering technology. Dashboard can be used as a user interface to Kubernetes. Kubernetes can build on Docker, but it is not required for running a Docker container environment.
You can run Kubernetes and Dashboard in Docker containers on the mainframe. For more information, see kubernetes.io

**Operations**

Docker setups often include components that monitor, control, or support operations across clusters.

**Elasticsearch, Logstash, Kibana (ELK)**

Elasticsearch, Logstash, and Kibana can run in containers on the mainframe, but you must build these components yourself for the z/Architecture. You can find sample Dockerfiles at [github.com/linux-on-ibm-z/dockerfile-examples/tree/master](http://github.com/linux-on-ibm-z/dockerfile-examples/tree/master).

For more information about ELK, see [elk-docker.readthedocs.io](http://elk-docker.readthedocs.io).

**Prometheus**

Prometheus is a community-maintained systems monitoring and alerting toolkit.

You can run Prometheus as a Docker image on the mainframe, but you must build it yourself for the z/Architecture. You can find a sample Dockerfile at [github.com/linux-on-ibm-z/dockerfile-examples/tree/master](http://github.com/linux-on-ibm-z/dockerfile-examples/tree/master).

For more information about Prometheus, see [prometheus.io/docs/introduction/overview](http://prometheus.io/docs/introduction/overview).

**etcd**

etcd is a community-maintained key/value store that can be used for secure inter-host communication.

A key/value store is an optional component, but might be required for some scenarios. For example, you can use it for overlay networks if Docker is not running in swarm mode. Other examples of popular key value stores include consul and zookeeper.

The distribution of your Docker host might include etcd. Alternatively, you can build it from the source on GitHub at [github.com/coreos/etcd](http://github.com/coreos/etcd).

You can run etcd in a container on the mainframe. For more information about etcd, see [coreos.com/etcd](http://coreos.com/etcd).

**Cloud integration**

Your Docker setup can be part of a cloud environment.

**IBM Cloud Private (ICP)**

ICP is a private cloud platform for developing and running workloads locally. It is an integrated environment that enables you to design, develop, deploy, and manage on-premises, containerized cloud applications behind your firewall.

For more information about ICP, see [www.ibm.com/cloud/private](http://www.ibm.com/cloud/private).

**IBM UrbanCode® Deploy (UCD)**

UCD manages source code and orchestrates the deployment of applications across environments. UCD can build on Docker in swarm mode or on Kubernetes, with or without ICP.
For more information about UCD, see [www.ibm.com/software/products/ucdep](http://www.ibm.com/software/products/ucdep)
Chapter 3. Planning for Docker

You must decide which workloads to run in containers and how to set up your Docker hosts.

Docker containers versus virtual machines

Depending on your workload needs, Docker containers or virtual machines can be the better choice.

Virtual machines are designed for isolation. Isolation is helpful for security and it helps to protect workloads from adverse effects of other workloads, for example, if another workload fails or competes for resources during heavy-load periods.

The Docker container technology is designed for fast deployment, resource sharing, and reuse of components. Sharing is conducive to a high workload density. Density here means the workload size that can be handled by particular hardware resources.

The following table contrasts some aspects of containers and virtual machines.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Containers</th>
<th>Virtual machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>The Docker host kernel has detailed information about the workload in containers.</td>
<td>The hypervisor has little information about the workload in virtual machines.</td>
</tr>
<tr>
<td></td>
<td>The Docker host kernel has container information at the operating system level, for example, about processes, files, or network connections.</td>
<td>The hypervisor has virtual machine information at the hardware level, for example about CPUs, memory, or I/O devices.</td>
</tr>
<tr>
<td>Isolation</td>
<td>Isolation is attained by actively hiding information, for example, with namespaces.</td>
<td>Isolation does not require much effort because there is not much information to hide. For example, the hypervisor does not have information about individual processes, and the virtual machine is only exposed to resources that are provided by the hypervisor.</td>
</tr>
<tr>
<td>Security</td>
<td>Malicious workloads can use a relatively broad interface of system calls for attacks.</td>
<td>Malicious workloads are constrained by a relatively narrow interface of hardware instructions that cause hypervisor intercepted.</td>
</tr>
<tr>
<td>Sharing</td>
<td>Sharing is possible because the Docker host can use its information about the containers. For example, containers can share libraries in the page cache (see &quot;Page cache sharing&quot; on page 14).</td>
<td>Sharing requires active discovery of information, for example, through Kernel Same page Merging (KSM).</td>
</tr>
</tbody>
</table>
Table 1. Containers versus virtual machines (continued)

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Containers</th>
<th>Virtual machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Density</td>
<td>Apart from the kernel, containers can share memory use of identical files, like software libraries and applications.</td>
<td>Virtual machines, typically, duplicate the kernel, software libraries, and applications. Duplication is a hindrance to a high workload density.</td>
</tr>
<tr>
<td>Speed of deployment</td>
<td>Container startup is swift because only the container context is loaded, not the entire operating system.</td>
<td>Virtual servers are started through an IPL, which includes booting Linux.</td>
</tr>
<tr>
<td>Operations</td>
<td>Basic Docker commands manage the entire container life-cycle: building, distributing, operating, and destroying containers.</td>
<td>Typically, managing the virtual server life-cycle spans components beyond the hypervisor.</td>
</tr>
</tbody>
</table>

Indicators for Docker

The following workload characteristics can serve as indicators for Docker as the hosting environment:

- Rapid deployment
- Frequent updates
- Microservices model, single application within a container
- Scaling by service
- DevOps
- Independence from systems programmer
- Docker skill

Accommodating sensitive workloads

You can isolate a container by running it alone on a dedicated host, for example, in a z/VM guest or in a KVM guest. This isolation is achieved at the expense of sharing potential and density, but you can run sensitive workloads within an existing Docker setup.

Obtaining the Docker product

The Docker product is available from different sources.

- IBM
- Docker Inc.
- Your distributor if your distribution includes a Docker package.

An important decision point for obtaining Docker is the available support.

Getting support

As an environment with components from multiple sources, you are likely to have more than one support organization for your Docker setup.
Disclaimer: Use the following table as guidance about who might offer support for various elements in a Docker environment. The table does not imply that support is available or that you are entitled to support. Support and its terms and conditions are subject to an agreement between you and the support provider.

Table 2. Support for elements in a Docker environment

<table>
<thead>
<tr>
<th>Support provider</th>
<th>Docker host operating system</th>
<th>Docker components</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributor</td>
<td>Red Hat Enterprise Linux SUSE Linux Enterprise Server Ubuntu Server</td>
<td>Docker components that the distribution includes as a package</td>
<td>Images that are included in the distribution. For example, the more recent support packs of SUSE Linux Enterprise Server 12 include an RPM with a base image for SUSE Linux Enterprise Server.</td>
</tr>
<tr>
<td>IBM</td>
<td></td>
<td>Docker Enterprise Edition for Linux on z Systems and LinuxONE (purchased from IBM)</td>
<td>adoptopenjdk/openjdk8 if IBM Java support has been acquired.</td>
</tr>
<tr>
<td>Docker Inc.</td>
<td></td>
<td></td>
<td>Numerous application images</td>
</tr>
<tr>
<td>Independent software vendor (ISV)</td>
<td>Linux for the mainframe distributions as provided by an ISV, for example ClefOS</td>
<td>Docker Community Edition (obtained from an ISV)</td>
<td>Images provided by an ISV, for example, a ClefOS image</td>
</tr>
<tr>
<td>No formal support, community-maintained.</td>
<td>Other distributions for the mainframe, for example: Fedora, OpenSUSE, Debian, Alpine</td>
<td>Docker Community Edition, Open Source Registry, ELK, Prometheus</td>
<td>Fedora image, OpenSUSE image, Debian image, Alpine image</td>
</tr>
</tbody>
</table>

Host setup

The resource requirements of your Docker hosts depend on the intended workload.

As a rule, give your Docker hosts the same resources that you would assign to a Linux instance for running the same workload without containers. You can pass on all storage devices, DASD or SCSI, and network interfaces to your containers.

Scaling with LPAR or virtual machines

The mainframe offers a highly scalable virtual hardware. The scaling options depend on whether you run your Docker hosts in LPAR mode or in virtual machines.

Running Docker hosts in LPAR mode can be advantageous for workloads that benefit from colocation or high density. For example, containers that are all instances of the same image and that work on a common queue of requests against the same database in the same partition are good candidates for running on a
Docker host in LPAR mode.

You can use the LPAR configuration to vertically scale your Docker host by assigning more resources to the LPAR. Within the host, you can then scale horizontally by creating more containers that process the workload in parallel. Running the same image in numerous containers on the same Docker host bears much potential for resource sharing (see "Page cache sharing" on page 14).

If you want to isolate tiers or tenants or to group resources for specific sets of containers, you might want to run your Docker hosts in virtual machines. The decision between z/VM or KVM is likely to be organizational rather than technical. If you already use z/VM at your installation, z/VM is a natural choice. The following graphic shows a z/VM hypervisor, but the scaling options are the same for Linux with KVM as the hypervisor.

In this setup, you can scale vertically at two different levels. You can scale the entire setup by assigning more resources to the LPAR. With the hypervisor, you can then individually scale each Docker host.

Within each Docker host, you can scale horizontally by starting more containers. You can also scale horizontally by creating more Docker hosts.

---

**Clustering options**

Run Docker hosts in swarm mode to use Docker’s clustering technology, or without swarm mode if you want to use Kubernetes.

Both technologies support deployment, updating, scaling, workload balancing, and enhanced availability for containers. Depending on the specific requirements.
your installation either technology might be more suitable. In comparisons, swarm
is usually deemed to be simpler and faster. Kubernetes is often considered more
flexible at the cost of complexity.

With both technologies, a cluster can include Docker hosts across hypervisors,
LPARs, mainframes, and Linux on hardware other than mainframes. Using
heterogeneous Docker hosts within a cluster can introduce constraints about which
workload can run on which host.

- For workloads with specific kernel requirements, you can identify eligible nodes
  with node labels.
- For deploying workloads across architectures, you can use multi-arch images. See
  “Multi-arch images” on page 22.

**Docker in swarm mode**

In swarm mode, Docker hosts are called nodes. Each swarm-controlled cluster
includes one or more swarm manager nodes and worker nodes on which the swarm
component acts as an agent.

Swarms usually include multiple swarm managers to enhance availability.
Manager tasks include cluster functions such as placement, scaling, and recovery
actions for the workloads.

The following figure illustrates a sample Docker installation on the mainframe.
Multiple Docker hosts, in swarm mode, run as guests of one z/VM hypervisor.
The swarm managers coordinate worker nodes with numerous workload
containers each. Registries, typically, serve the entire enterprise rather than a
particular swarm. In the example, the registry runs on a Docker host on the same
hypervisor, but it is not part of the swarm.

![Figure 5. Layout of a Docker installation with swarm](image)

This sample installation could also use KVM guests instead of z/VM guests, or
each Docker host could run on a separate LPAR.

**Kubernetes**

With Kubernetes, you have masters that manage clusters of nodes. Nodes are Docker
hosts with a Kubernetes agent that provides an API for Kubernetes masters.

Each node can contain one or more pods. A pod contains one container or multiple,
usually tightly coupled, containers.
Functional groups of pods can be assigned to a service. External applications access the pods and containers through services. By deploying, replicating, scaling, and updating pods, masters manage services according to predefined goals in a policy. Services and policies are specified in a descriptive configuration.

Kubernetes clusters usually include multiple masters to enhance availability.

This sample installation could also use KVM guests instead of z/VM guests, or each Docker host could run on a separate LPAR. The registry is not usually part of a regular cluster.

For more information, see [kubernetes.io](https://kubernetes.io).

### Page cache sharing

Memory-sharing potential arises from a Docker host with multiple containers that are created from the same image or from images with shared ancestors.

With a suitable setup, only one copy of the shared container content needs to be in memory. The best effect is attained if frequently used code, for example library routines, can be run from the shared Linux page cache.

The IBM mainframe was designed and optimized to run large operating system instances with heavy workloads:

- It is renown for the reliability that is essential for running large workloads for which a failure would have a high business impact.
- It provides large memories and an efficient CPU-cache structure, so that it can exploit the memory sharing potential of images to realize high-density Docker hosts with numerous containers.

The shared Linux page cache can be seen as a staging area for the mainframe CPU-caches. Sharing at the page cache level has prerequisites:

- The Docker host must use a storage driver that supports page cache sharing.
- The containers must share a suitable image layer stack.
Storage drivers

Docker *storage drivers*, also called *graph drivers*, manage images, containers, and their layers on a Docker host.

Docker supports several storage drivers with different capabilities, technologies, and Linux distribution affinities, see [docs.docker.com/engine/userguide/storagedriver/selectadriver](https://docs.docker.com/engine/userguide/storagedriver/selectadriver).

Within a Linux file system, files are unambiguously described by *inode* numbers. Through multiple inode structures that contain the same inode number, different paths in a file system can be used to access the same file. When multiple processes access a file through the same inode number, the Linux kernel maintains only one copy of the file in the page cache.

Docker storage drivers can support page cache sharing by using the same inode number for equivalent files across containers.

Most distributions include *devicemapper* as a possible storage driver for Docker. However, it works at the block device level and does not support page cache sharing. Because container file systems map to different block devices, Linux cannot trace the device mapper devices back to a common inode number.

The following storage drivers support page cache sharing. They all work at the file system layer and use union mount.

**overlay**

as of kernel 3.18, *overlay* is included in the mainstream Linux kernel. By design, it uses inodes liberally, which can limit the number of images that the Docker host can support, especially for containers with numerous small files.

*Tip:* You configure the number of available inodes when you format a file system.

**overlay2**

as of kernel 4.0 you can use *overlay2*. This storage driver uses inodes more efficiently than *overlay*. With *overlay2*, containers are limited to about 120 layers.

*Tip:* You can keep the number of layers down by running multiple commands and by adding multiple files with a single RUN or ADD instruction in your Dockerfiles. As of the docker engine API version 1.25, you can also use the --squash option of the `docker image build` command to merge multiple layers.

**aufs**

is the first storage driver that was used by Docker. It is not part of the mainstream Linux kernel, and not all distributions include it.

See [docs.docker.com/storage/storagedriver/select-storage-driver](https://docs.docker.com/storage/storagedriver/select-storage-driver) for an overview of which storage driver is supported by which distribution.

Designing images for sharing

Containers that use the same base image can share files in the base image layer. Files in subsequent image layers can be shared only if these layers have the same content and are added in the same sequence to the stack of image layers.
The following figure illustrates the sharing potential for three images that are all derived from the same base image. Two libraries, libA and libB, are added in consecutive image layers. For the first two images libA is followed by libB, in the third image the order is reversed. As a result, the first and second images can share the libraries. The third image can share only files from the base image with the other two.

Image layers are defined by their content and by their parent image. For the first and second image in Figure 7, the libB layer has the libA layer with ID 314a7... as the parent. In the third image, the parent of libB is the base image layer, with ID ba125... Because the parent images have different IDs, libA in the third image and all its descendant layers have different IDs from their counterparts in the first and second images, regardless of common content.

Sharing is possible only across image layers with matching IDs. Matching image layer IDs layers necessitate a common line of parent images down to a common base image.

**Tips:**
- For a particular host, use a limited number of base images.
- Design your images such that layers with extensive common content are added first to the base image.
- Design your images such that you have a large common layer stack from which variants are derived in the top layers.
- Generally, use as much common content as possible in ancestors low down the layer stack.

**Confirming shared content**

The following scenario assumes a large image, shared_image, with an executable file, /usr/bin/big_binary, that contains 256 MB of code and read-only data. The scenario also assumes that the storage driver supports page cache sharing. The `free` command is issued before and after instances of shared_image are started as containers.

The command output of the example shows that starting a container leads to an increase of about 256 MiB in the page cache. Accordingly, the amount of free memory is lower.
The memory status after a second container is started confirms that memory is being shared. Running the second container does not significantly increase the page cache and the amount of free memory remains the same within the accuracy of the measurement.

With this sharing characteristic, the more containers you run, the more notable are the memory savings.
Chapter 4. Managing images

You must find suitable images, adapt them where necessary, keep them current, and dispose of them when they are no longer needed.

Acquiring images

Docker on IBM Z and LinuxONE requires images that support the z/Architecture.

You have several options for obtaining such images:
- Use an existing image.
- Extend an existing image.
- Create an image from scratch.

Using existing images

Docker store provides a multitude of images for various architectures, including z/Architecture.

Docker store does not enforce a naming scheme with respect to image architecture. However, image providers usually use “s390x” as suffixes to the for images that are compiled for z/Architecture. Search Docker store for “s390x” to find these images. Image names with a leading “s390x/” are provided by developers from IBM and Docker Inc.

You can also use multi-arch images that include s390x support. Generally, images that are provided by Docker Inc. support multiple architectures, including s390x.

For more information about multi-arch images, see “Multi-arch images” on page 22.

The image description usually states the image architecture or architectures and provides information about the maintainer, source base, variants, stability, and support for the image.

If searches on Docker store do not yield a result, the search is expanded automatically to include Docker hub. Although Docker hub is maintained by Docker Inc., images on Docker hub are less scrutinized by Docker Inc. than images on Docker store.

You can access Docker store at store.docker.com.


Extending existing images

You can use a Dockerfile to specify instructions for extending an existing image.

Typically, these instructions perform the following tasks:
1. Get an existing z/Architecture image, for example, an Ubuntu base image or a Java image.
2. Install upgrades and software.
3. Define network and volume requirements.
4. Specify a command to run when the image is used to start a container.

The following simplified example shows the main instructions in a Dockerfile.

```bash
FROM s390x/ubuntu:16.04
MAINTAINER Whatever my name is <some@address.com>

# run commands:
RUN apt-get update && apt-get install -y apache2

# copy files into the image
ADD index.html /var/www/html/

# publish a port of the container
EXPOSE 80

# how the container is started
ENTRYPOINT ["/usr/sbin/apachectl","-DFOREGROUND"]
```

FROM refers to the parent image, which is the starting point for the new image. This starting point can be a base image or an entire image stack with a base image as the bottom layer. The image that results from running the Dockerfile inherits the entire layer stack of the parent image and has one or more new layers at the top of the stack.

The Dockerfile of the example refers to an Ubuntu image for the z/Architecture. The example also shows a tag, 16.04, which specifies the 16.04 version of the image.

**Terminology note:** Do not confuse a parent image with a base image, which is the bottom layer of an image stack. A parent image can consist of multiple layers. Some sources mistakenly call a parent image the base image of a Dockerfile.

MAINTAINER specifies the owner of the new image, usually the author of the Dockerfile.

RUN runs one or more commands to modify the image, for example by installing extra packages.

**Tip:** Each RUN instruction results in a separate image layer. To curb layer sprawl, specify several commands with one instruction. As of the docker engine API version 1.25, you can also use the --squash option of the docker image build command to merge multiple layers.

In the example, one RUN instruction first updates Ubuntu and then installs the apache2 package.

ADD adds files or directories to the image file system.

**Tip:** Each ADD instruction results in a separate image layer. To curb layer sprawl, add several items with one instruction. As of the docker engine API version 1.25, you can also use the --squash option of the docker image build command to merge multiple layers.
In the example, the ADD instruction copies a file, `index.html`, from the directory that contains the Dockerfile to `/var/www/html` in the image root file system.

**EXPOSE**
sets up a port on which containers that are created from the new image can listen for connections.

**ENTRYPOINT**
runs a command when the image is used to start a container. In the example, the control interface of an Apache server is started.

For the Dockerfile syntax and best practice advice, go to the following web pages:
- docs.docker.com/engine/reference/builder
- docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices
- crosbymichael.com/dockerfile-best-practices.html

### Base images for the Linux for Z distributions

A likely starting point for extending an existing image is a base image for a Linux for Z distribution.

#### Officially supported distributions for the mainframe

**SUSE Linux Enterprise Server**
SUSE Linux Enterprise Server 12 includes an RPM with a base image for the z/Architecture.

**Ubuntu Server**
Docker Inc. provides a multi-arch base-image for Ubuntu Server on Docker Store.

**Red Hat Enterprise Linux**
Red Hat provides base images for Red Hat Enterprise Linux at access.redhat.com/containers.

#### Community-maintained distributions

**openSUSE**
You can find a z/Architecture base image for openSUSE on Docker hub at store.docker.com/community/images/s390x/opensuse

**Debian**
You can find a multi-arch base-image for Debian on Docker Store at store.docker.com/images/debian

**ClefOS**
You can find a z/Architecture base-image for ClefOS on Docker hub at store.docker.com/community/images/sinenomine/clefos-base-s390x

ClefOS is based on CentOS, which is based on Red Hat Enterprise Linux.

**Alpine**
Docker Inc. provides a multi-arch base-image for Alpine on Docker Store.
This slim image has a size of only 5 MB.
Creating an image from scratch

A slim purpose-build base image that provides no more than what is required can boost speed of deployment, workload density, and the overall performance of your installation.

Before you begin: Creating Docker images from scratch is an expert task that requires advanced Linux skills. A detailed description is beyond the scope of this publication.

The following steps are intended to convey the general idea of how to proceed:
1. Set up a chroot-like branch of your file system by copying all system files that are required by the new image into that branch.
2. Install any required packages to the new branch.
3. Remove all superfluous content.
4. Create a .tar file that contains the file system branch.
5. Import the .tar file into Docker.

For an example that imports a .tar file with the docker import command, go to containerz.blogspot.com/2015/03/creating-base-images.html

Alternatively, you can build a base image from a Dockerfile that specifies the scratch keyword instead of a parent image. For an example, go to docs.docker.com/engine/userguide/eng-image/baseimages/#creating-a-simple-base-image-using-scratch

Multi-arch images

Multi-arch images support workload balancing within Docker-host clusters that span architectures.

A multi-arch image references multiple existing images that are compiled for different architectures. When a container is started from a multi-arch image, Docker selects and uses the image that matches the architecture of the Docker host.

For information about creating multi-arch images, see containerz.blogspot.com/2016/07/multi-arch-registry.html

For examples of multi-arch images, see the images that Docker Inc. provides on Docker store at store.docker.com

Keeping images current

As for any software, container currency is desirable to use the latest features of an application and sometimes necessary to apply security patches.

You probably have two types of images to keep current:
• Images that you pull from a registry.
• Images that you have extended by using a Dockerfile.

Updating images from a registry

You can update an image by pulling it from a registry with an appropriate tag for an updated version. Images often use latest as an alias tag that maps to the latest version. If you omit the tag, latest is used by default.
The updated image then has the original image name but a new image ID, and the old image is retained with its original ID. Thus, the new image can be rolled out gradually to containers and a rollback to the original image is possible.

**Updating extended images**

If a layer in a parent image or any other layer of an image has been updated, the image must be rebuilt from its Dockerfile to include the update.

If a tag is used, updates are installed with the current version according to the tag. Omitting the tag or specifying `latest` updates the image to the latest version.

As for updated images that are pulled from a registry, the result is two images. The updated image is available with the original name but with a new ID. The original image is retained and available with the original ID.

To cascade an update through a succession of derived images all images that contain a changed layer must be rebuilt.

**Updating containers**

The updates to an image are propagated to a container when the container is restarted. In swarm mode, you can configure rolling updates to containers that are based on older versions of an image within the swarm.
Chapter 5. Security

Docker adds new entities to a mainframe setup: images, containers, and Docker hosts. These entities must be considered for their security impact.

Image security

As with any software, you must decide which image source to trust in terms of security, reliability, and pedigree.

Maintaining a private registry is one way to control which images can be deployed in your environment. However, unless you create all your images yourself, you have to import images to this private registry.

You can use the Docker Content Trust and Notary functions to assure image authenticity. Docker Content Trust and Notary prevent images from being processed unless they are signed by their provider and the signature can be verified. For example, you can prevent unsigned images from being pulled from a registry.

The community-based open registry is also evolving fast to include various security features.

Docker images have the advantage of being relatively easy to update. Security fixes for a base image or an application layer can easily be propagated to derived images that include them as layers.

Good practice rules:

• Use the same due diligence about vetting images that you would use for vetting other software.
• Whenever possible, use Docker certified images, official images, or images that are signed by trusted sources.

Container security

The main concern about container security is the level of isolation among containers and between containers and the host operating system.

In general, Docker containers share more resources than virtual machines, most notably the host's operating system kernel. Sharing the kernel can boost the efficiency of Docker containers but you must decide which workloads can be allowed to coexist on the same Docker host.

Docker does not necessarily weaken security. If the containers on a Docker host run workloads that would otherwise run as applications on the same Linux instance the opposite is true: Docker adds a layer of isolation.

Docker uses established Linux features to isolate containers from one another.

namespaces

Docker uses namespaces to separate host and domain names, interprocess communications, process IDs, network settings, and mount points.
By default, the mount point namespace limits the root file system, as seen by a container, to the image root file system. An image root file system is just a branch of the host's file system. Processes that run in the container cannot detect files beyond this scope.

Optionally, the user namespace can remap container users, including root (ID zero), to different users on the Docker host.

The container root user is constrained by capabilities, even if it is identical with the root user of the host system.

capabilities
Docker uses capabilities to restrict the system calls that a container application is permitted to use in the container context.

Tip: The default capabilities that are used by Docker are suitable for most purposes. With expert knowledge of your container workload, you can use the Docker `--cap-add` and `--cap-drop` options to fine-tune the capabilities when starting or creating a container.

control groups
Docker uses control groups (cgroups) to manage host system resources for containers, such as CPU, memory, and I/O. With cgroups, Docker can prevent a container from using too many resources at the expense of other containers.

Processes from different containers run independently from one another and cannot interfere.

seccomp
Docker provides a default profile that permits the system calls that are needed to run the Docker container but is restrictive otherwise. For more information and sample profiles, see the Docker seccomp page.

AppArmor
In addition to the Docker security features, you can deploy AppArmor to further curb container privileges.

Docker generates a default AppArmor profile, which is a good starting point. For more information and sample profiles, see the Docker engine security/apparmor page.

SELinux
In addition to the Docker security features, you can deploy SELinux to further curb container privileges. A good starting point for an SELinux profile for your container is the profile that you would use for the application if it were to run outside the container context.

By default, containers are limited in scope and authority. You can deliberately override the default security features and set up privileged containers with broad access to the host's root file system and ample privileges. For example, you might want to use a container to run administrative tasks. Override security features only if you can understand and control the vulnerabilities that you incur.

Docker provides a tool, docker container diff, to assess the changes that a container makes to its file system. This tool can help you detect suspicious behavior of containers. For more information, see the Docker engine reference/commandline/container_diff page.
Containers deliberately share host resources. As a consequence, some of these resources are visible from the container context. For example, the `free` command shows information about the host's entire memory, and the `dmesg` command shows all kernel messages of the host. Sharing the kernel also means sharing cryptographic keys and certificates.

**Good practice rules:**
- Do not give containers more access to the host root file system than necessary.
- Ensure that the security features of your choice are set up and enabled, for example, AppArmor or SELinux.
- Run containers with sensitive workloads alone on a dedicated Docker host.
- Write container logs to persistent storage.

**Docker host**

Like a hypervisor, a Docker host is a sensitive control point for the workloads that it hosts.

Docker uses capabilities. Use a Linux security feature like AppArmor or SELinux to further harden the host operating system.

In swarm mode, Docker sets up a cluster CA and uses Transport Layer Security (TLS) for mutual authentication among nodes. Communications between nodes is encrypted.

**Good practice rules:**
- Use a hardened Linux instance with no unnecessary users, access methods, and ports.
- Use a Docker host exclusively for Docker and its containers.
- Do not expose the REST API to unauthorized users, external (HTTP) or internal. In particular, do not lightly add users to the docker user group. Users in this group can gain root privileges.

**Mainframe cryptographic features**

Containers can be set up to use the mainframe's cryptographic hardware features, specifically, CPACF and Crypto Express.

**CPACF**

CP Assist for Cryptographic Function (CPACF) provides hardware instructions for cryptographic algorithms that are run on the core.

If the Docker host has access to CPACF, so do its Docker containers. No special configuration is needed.

All software running in the containers can use the CPACF machine instructions in the same way as if running directly on the host.

CPACF supports clear key symmetric cryptography for applications that build on libica or use the hardware instructions directly.
Crypto Express adapters

Cryptographic adapters are available as hardware features, Crypto Express features, that Docker containers can access through a device node and an associated device driver in the kernel.

Unless the base image already includes the /dev/z90crypt device node, create it with the following option of the `docker run` command:

```
--device /dev/z90crypt
```

Through this node, the container can access all cryptographic adapters that are available to the Docker host. Depending on its configuration, a cryptographic adapter offers different capabilities:

**Accelerator**

- Clear key support only.

**CCA coprocessor**

- Clear key, secure key, and protected key support; random numbers through the /dev/hwrng device node.

**EP11 coprocessor**

- Secure key support through a PKCS#11 API only.

For information about key types, see "Cryptography with secure and protected keys" on page 31.

Installing and configuring cryptographic adapters is beyond the scope of this publication. The sections that follow assume that an adapter is available to the Docker host and is configured as an accelerator, or as a CCA or EP11 coprocessor.

Use the request counters of the `lszcrypt` command output to confirm that your containers use the cryptographic adapters. When evaluating your results, be aware that `lszcrypt` is not limited to the container scope, but provides statistics across the Linux instance. Depending on your distribution, the package that provides `lszcrypt` is `s390-tools` or `s390utils`.

**Example:**

```
# lszcrypt
CARD.DOMAIN TYPE     MODE     STATUS  REQUEST_CNT
-------------------------------
05    CEXSC CCA-Coproc online 904
05.0038 CEXSC CCA-Coproc online 904
06    CEXSP EP11-Coproc online 0
06.0038 CEXSP EP11-Coproc online 0
0a    CEXSA Accelerator online 22
0a.0038 CEXSA Accelerator online 22
```

Cryptographic adapters are subdivided into multiple domains. Each domain has its own state, including its own master key.

In the `lszcrypt` output, adapters are identified by two-digit hexadecimal device numbers. Domains are identified by four-digit hexadecimal numbers. The combination `<adapter_id>,<domain_id>` identifies a specific domain on a specific adapter and is also called an AP queue. AP queues that hold identical master keys can be used for redundancy and workload balancing.
The `lszcrypt` sample output shows three adapters, with IDs 05, 06, and 0a. Each adapter provides the domain with ID 0038. Thus, three different AP queues are available for cryptographic operations: 05.0038 in CCA coprocessor mode, 06.0038 in EP11 mode, and 0a.0038 in accelerator mode.

**Software stack**

Your Docker image must contain all user space software components that you would also need for your application to run on a separate Linux instance.

Figure 8 illustrates the components that interface with the zcrypt device driver and with CP ACF.

**openCryptoki**

Applications can access cryptographic adapters through openCryptoki, an open source implementation of the Cryptoki API.

**Required packages**

Install the openCryptoki library that is provided with your distribution. The tokens that are required for a particular adapter configuration are available with the package. If you want to access an accelerator though libica, also install the `libica` package of your distribution.

For EP11 or CCA, you also have to install the `libep11` or `csulcca` package. You can obtain these packages from IBM at [www.ibm.com/security/cryptocards/pciecc/zlinuxsoftware.shtml](http://www.ibm.com/security/cryptocards/pciecc/zlinuxsoftware.shtml)
Configuration

openCryptoki depends on a running pkcsslotd daemon. You can use a startup shell script for the Docker image to assure that the daemon is started before your cryptographic application.

Example:

```bash
#!/bin/bash

# First start slot daemon
/usr/sbin/pkcsslotd

# Then start my application that uses openCryptoki
/bin/crypto-app
```

openCryptoki stores key related information at /var/lib/opencryptoki. To maintain a persistent key configuration for Docker containers, use the --volume option `docker run` command to provide a persistent volume for this directory.

The initial setup of openCryptoki, for example the token initialization and setting the user PIN, can be done in advance for the image, or it can be done interactively when starting the container the first time.

Example for starting a container

Assuming that the openCryptoki configuration for your container instance is stored at /data/opencryptoki/crypto-app/0001 in the host file system, that the image name is crypto-app, and that a shell script `start-crypto-app.bash` for starting the pkcsslotd daemon and the application is in place within the image, start the container with the following `docker run` command:

```bash
# docker run -ti --rm 
-v /data/opencryptoki/crypto-app/0001:/var/lib/opencryptoki 
--device=/dev/z90crypt crypto-app start-crypto-app.bash
```

Clear key asymmetric cryptography

Containers can use mainframe cryptographic adapters to accelerate asymmetric clear key cryptography. In clear key cryptography, keys are stored unencrypted in the Linux memory.

Note: For performance reasons, clear key symmetric cryptography is typically performed using CPACF, see “CPACF” on page 27.

For clear key acceleration, you can either use libica or you can use openCryptoki with the ICA token (see Figure 8 on page 29). You can set up and use your image in the same way as you would configure your application on the host.

You have to pass /dev/z90crypt to the containers with the --device option of the `docker run` command. Containers can use this device to access all cryptographic adapters that are accessible by the host and that are configured for clear key encryption. From a security point of view, running multiple such containers is akin to running several applications in parallel on the host.
Cryptography with secure and protected keys

Containers can use mainframe cryptographic adapters in CCA mode and in EP11 mode for cryptographic operations with encrypted keys, and as of CCA 6.0, also with protected keys.

The following key types are relevant to cryptography on Z:

**Clear key**
Keys are stored unencrypted in the Linux memory.

**Master key**
A key that is stored on a cryptographic adapter and is used for creating secure keys.

**Secure key**
A key that is generated on a cryptographic adapter and encrypted with a master key.

**Virtual server master key**
A key that is generated by a hypervisor and that is not accessible by the operating system that runs in the virtual server. The hypervisor can be PR/SM™, z/VM, or KVM on Z.

**Protected key**
A key that is encrypted with the virtual server master key and can be used by the CPACF of the virtual server. A new virtual server master key is created each time that a virtual server is started or stopped.

For secure keys, all cryptographic operations of a container must be based on the same master key. If a container is configured to access multiple AP queues, these AP queues must hold the same master key.

If you build an image with a hardcoded AP queue configuration, all derived containers use this configuration.

For more flexibility, you can build images for which the AP queues are configured with the `docker run` command when the container is started. The techniques that you can use depend on the coprocessor mode, CCA or EP11.

Regardless of the AP queue configuration with which a container is started, containers can gain access to all AP queues of the Docker host. The techniques for starting a container with a suitable selection of AP queues are a technical convenience, not a means of isolation. As for any container workloads, run containers on the same Docker host only if you would also run these workloads directly on the same operating system instance.

**CCA as of version 6.0**
Use the `CSU_DEFAULT_ADAPTER` and `CSU_DEFAULT_DOMAIN` environment variables to specify a single AP queue for your container.

You pass the variables to the container with the `--env` parameter of the `docker run` command.

**Example:** `--env CSU_DEFAULT_ADAPTER=CRP11 --env CSU_DEFAULT_DOMAIN=56`

Both environment variables specify decimal values.
CSU_DEF_ADAPTER

specifies CRP<nn> where <nn> equals the device number of the

cryptographic adapter plus 1. For example, CRP11 specifies the card with
device number 10.

CSU_DEF_DOMAIN

specifies a domain number.

In contrast to these environment variables, the output of the lszcrypt command
shows card numbers and domain specifications in hexadecimal notation. Thus, the
environment variables of the example refer to AP queue 0a.0038 in the lszcrypt
output of “Crypto Express adapters” on page 28.

For details about the CCA configuration, see Secure Key Solution with the Common

EP11

For EP11, you can configure the AP queues with the /etc/opencryptoki/
ep11tok.conf file.

Example: To specify two AP queues, domain 56 on adapter 6 and domain 56 on
adapter 10, you can create a file ep11tok.conf with the following contents:

APQN_WHITELIST
06 56
10 56
END

For more details about the EP11 configuration, see Exploiting Enterprise PKCS #11
using openCryptoki, SC34-2713.

To start a container with a custom AP queue configuration you can use one of the
following techniques:

• Create custom ep11tok.conf files on the host and use --volume option of the
docker run command to mount them into the containers.

• Create custom ep11tok.conf files in the Docker image and select them with the

• From a startup script, create a ep11tok.conf file that uses environment variables
for one or more AP queues, then pass values for the variables with the --env
option of the docker run command.

Multiple ep11tok.conf files on the host

On the host, create a separate ep11tok.conf file for each required container
configuration.

Example:

```bash
# cat /data/ep11tok/ep11tok-06_10-56.conf
APQN_WHITELIST
06 56
10 56
END

# cat /data/ep11tok/ep11tok-06_10-57.conf
APQN_WHITELIST
06 57
10 57
END
```
Select the AP queue configuration with the `--volume` option of the `docker run` command.

Example:

```bash
# docker run --device /dev/z90crypt \
   -v /data/ep11tok/ep11tok-06_10-56.conf:/etc/opencryptoki/ep11tok.conf \
   crypto-app start-crypto-app.bash  \
...  
# docker run --device /dev/z90crypt \
   -v /data/ep11tok/ep11tok-06_10-57.conf:/etc/opencryptoki/ep11tok.conf \
   crypto-app start-crypto-app.bash  
```

Multiple ep11tok.conf files in the docker image

If the environment variable `OCK_EP11_TOKEN_DIR` is set, the EP11 token looks for the `ep11tok.conf` file in the specified directory.

You can build a Docker image with preconfigured `ep11tok.conf` files in different directories.

Example:

```bash
# cd /ep11configs 
# find -name ep11tok.conf 
./ep11-06_10-56/ep11tok.conf 
./ep11-06_10-57/ep11tok.conf 
```

You can then select the AP queue configuration by specifying a value for the `OCK_EP11_TOKEN_DIR` environment variable with the `--env` option of the `docker run` command.

Example:

```bash
# docker run --device /dev/z90crypt \ 
   --env OCK_EP11_TOKEN_DIR="/ep11configs/ep11-06_10-56" \ 
   crypto-app start-crypto-app.bash  \
...  
# docker run --device /dev/z90crypt \ 
   --env OCK_EP11_TOKEN_DIR="/ep11configs/ep11-06_10-57" \ 
   crypto-app start-crypto-app.bash  
```

Startup script to create the ep11tok.conf file

Extend your startup bash script to create an `ep11tok.conf` file with one or more environment variables, one for each AP queue.

Example:

```bash
#!/bin/bash

# Create ep11tok.conf file
cat > /etc/opencryptoki/ep11tok.conf << EndEP11TOK
APQN_WHITELIST
$APQN_1
$APQN_2
END
EndEP11TOK
```
# Start slot daemon
/usr/sbin/pkcsslotd

# Then start my application that uses openCryptoki
/bin/crypto-app

You can then select the AP queues with one or more --env options of the `docker run` command.

Example:

```
# docker run --device /dev/z90crypt \  
--env APQN_1="06 56" --env APQN_2="10 56" \  
crypto-app start-crypto-app.bash \  
...# docker run --device /dev/z90crypt \  
--env APQN_1="06 57" --env APQN_2="10 57" \  
crypto-app start-crypto-app.bash  
...`
```

**Random data**

Containers need a device node to access hardware-supported random data.

Unless your base image already includes this node, create it with the following option of the `docker run` command: --device /dev/prandom

Use the same technique for other device nodes that provide random data.
Chapter 6. Avoiding common pitfalls

How to avoid common problems.

Use applicable images

Use images that are compiled for z/Architecture and that do not require a more recent Linux kernel or a more recent mainframe model than what is provided by the Docker host.

No image layer must require kernel features that are not provided by the Linux instance of the Docker host.

All layers in an image must be compiled for z/Architecture. Otherwise, container start commands with the image fail with an obscure error message like this example:

```
standard_init_linux.go:178: exec user process caused "exec format error"
```

Docker does not require a particular mainframe model, but images can have hidden hardware dependencies. For example, if an image layer is compiled to require IBM z13® it cannot be used for running a container on IBM zEnterprise® EC12 (zEC12).

Avoid remote container access

An ssh connection to a container can be convenient for debugging but introduces an unnecessary vulnerability.

Do not provide remote access to your container unless the application itself requires this access. A better alternative to an ssh port would be to start an additional shell in the container, for example, with a command of this form:

```
# docker exec -it <id> bash
```

where <id> is the container ID.

Use a host kernel with Docker support

The Linux kernel of the Docker host must be compiled to support all features that are required by the Docker engine.

**Kernel builders:** This information is intended for those who want to build their own kernel. Be aware that both compiling your own kernel or recompiling an existing distribution usually means that you have to maintain your kernel yourself.

Among other requirements, the Linux instance of a Docker host needs the following features:

- namespaces and cgroups for container isolation
- unification file systems for image and container implementation

The required kernel configuration options have complex dependencies. Go to GitHub at [github.com/moby/moby/blob/master/contrib/check-config.sh](https://github.com/moby/moby/blob/master/contrib/check-config.sh) for a
script that checks a Linux instance for the Docker requirements.
Accessibility

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

Documentation accessibility

The Linux on Z and LinuxONE publications are in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when you use the PDF file and want to request a Web-based format for this publication, use the Readers' Comments form in the back of this publication, send an email to eservdoc@de.ibm.com, or write to:

IBM Deutschland Research & Development GmbH
Information Development
Department 3282
Schoenaicher Strasse 220
71032 Boeblingen
Germany

In the request, be sure to include the publication number and title.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility at

www.ibm.com/able
Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.
This information is for planning purposes only. The information herein is subject to change before the products described become available.

**Trademarks**

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml)

Adobe is either a registered trademark or trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Index

Special characters
--cap-add, option 25
--cap-drop, option 25
--device, option 27, 30, 34

A
accessibility 37
ADD instruction, Dockerfile 19
Alpine 21
API, Docker 5
AppArmor 25
aufs 15

B
base image
Alpine 21
ClefOS 21
concept 2
Debian 21
openSUSE 21
Red Hat Enterprise Linux 21
SUSE Linux Enterprise Server 21
Ubuntu Server 21
Buying 10

C
cache, sharing 14
capabilities 25
CCA 31, 32
CentOS 21
groups 25
chroot() 1
ClefOS 21
CLI, Docker 5
cluster management 6
clustering 12
command
docker container diff 25
docker import 22
docker inspect 19
container basics 2
security 25
update 22
versus virtual machine 9
containerd, daemon 5
control groups 25
CPACF 27, 30, 34
CPU cache 14
Crypto Express 28
cryptographic adapters 27, 30, 34

D
Dashboard 6
DCT 25
Debian 21
density, workload 9
devicemapper 15
distribution, base image 21
Docker 10
CLI and API 5
components 5
core 2
e1 1
e2 5
host 1
image 2
support 11
docker container diff command 25
Docker Content Trust (DCT) 25
Docker Hub 6
docker import command 22
docker inspect command 19
Docker Store 6
Docker Trusted Registry (DTR) 6
docker user group 27
Dockerfile 19
driver, storage 15
DTR 6
E
Elasticsearch, Logstash, Kibana 7
ELK 7
eengine, Docker 1
ENTRYPOINT instruction, Dockerfile 19
EP1 31, 32
etcd 7
EXPOSE instruction, Dockerfile 19
F
FROM instruction, Dockerfile 19
G
graph driver 15
guest versus LPAR 11
H
horizontal scaling 11
host, Docker 1
hypervisor
versus Docker host 9
z/VM and KVM 11
I
IBM Java 19
IBM UrbanCode Deploy (UCD) 7
image basics 2
design for sharing 10
image (continued)
finding 19
multi-arch 22
requirements 35
security 25
update 22
node 15
isolation versus sharing 9
J
Java 19
K
k8s 13
Kibana 7
Kubernetes 6, 13
KVM 11
L
label, node 13
latest, tag 22
layer
container 2
image 2
sharing 10
local storage area 3
logging 7
Logstash 7
LPAR versus virtual machine 11
M
manager node, swarm 13
memory, sharing 14
microservices 2
monitoring 7
multi-arch images 22
multi-container application 6
N
namespaces 25
node, swarm 13
Notary 6
O
Open Source Registry 6
openCryptoki 29
openSUSE 21
orchestration 6, 12
overlay 15

© Copyright IBM Corp. 2017, 2018
Readers’ Comments — We'd Like to Hear from You

Linux on Z and LinuxONE
Running Docker Containers
on IBM Z
October 2018

Publication No. SC34-2781-01

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.
Submit your comments using one of these channels:
• Send your comments to the address on the reverse side of this form.
• Send your comments via email to: eservdoc@de.ibm.com

If you would like a response from IBM, please fill in the following information:

Name_________________________________________Address______________________________

Company or Organization______________________________

Phone No.________________________________________Email address_______________________
Readers’ Comments — We’d Like to Hear from You
SC34-2781-01

IBM Deutschland Research & Development GmbH
Information Development
Department 3282
Schoenaicher Strasse 220
71032 Boeblingen
Germany

PLACE POSTAGE STAMP HERE