OCI Image Format

Glyn Normington

May 28, 2019

This document provides a formal model of some aspects of the *OCI image format*. It describes the objects which make up an OCI repository and how content addressing is used to refer to these objects.

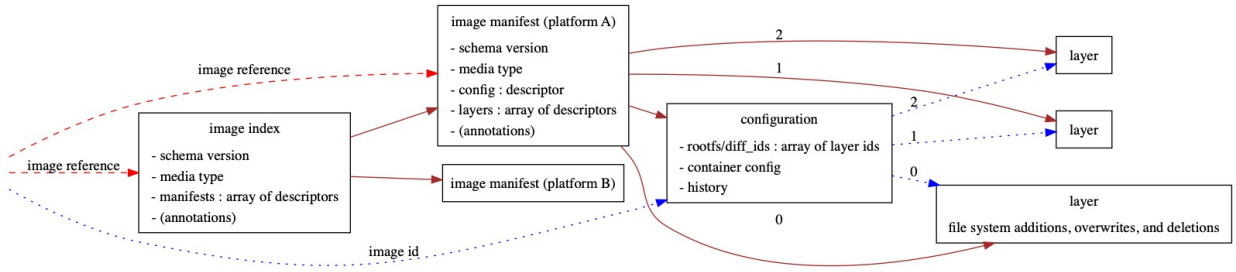
Contents

1 Introduction

Docker Inc. ([1]) introduced *container images* and *registries* to hold them and these were later standardised as part of the Open Container Initiative ([2]). The reader is assumed to have a basic understanding of how images are used.

Registries are named collections of *repositories*. The relationship is described formally in *Image Registries* ([3]).

This document models how image are represented and stored in repositories. This is based on the *OCI Image Format Specification* ([4]). It does not cover all details, but aims to provide an overview of how images are structured as layers and gathered into a repository. The following diagram shows the main objects that are modelled.



The Z specification language is used to describe the model, but sufficient English text is also provided that readers who do not know Z should be able to get a reasonable idea of what's going on. The appendix contains a summary of the Z notation. For more information about Z, please consult the Z Manual ([5]). The model was type checked using `fuzz` ([6]).

2 Content Addressing

Before we describe the objects which make up a repository, we model how objects refers to each other using *content addressing*. In content addressing, a collision resistant function[1] is applied to the content of an object to produce a content address which can be used to refer to the object.

OCI uses a form of *content address*, known as a *digest*, which is a cryptographic hash of the *compressed* content of an object.

$$[Digest]$$

OCI also uses another form of content address, known as an *id*, which is a cryptographic hash of the *uncompressed* content of an object.

$$[Id]$$

Content addressable storage (CAS) in a repository has a collection of objects uniquely identified by digest and, sometimes, by id. Each object has a "raw" (possibly compressed) size in bytes.

$$\begin{array}{|l} \textit{CAS}[\textit{Object}] \\ \hline \textit{deref} : \textit{Digest} \rightarrowtail\!\!\!\!\!\!\rightarrow \textit{Object} \\ \textit{identify} : \textit{Id} \rightarrowtail\!\!\!\!\!\!\rightarrow \textit{Object} \\ \textit{size} : \textit{Digest} \pfun \mathbb{N} \\ \hline \operatorname{ran} \textit{identify} \subseteq \operatorname{ran} \textit{deref} \\ \operatorname{dom} \textit{deref} = \operatorname{dom} \textit{size} \\ \hline \end{array}$$

As well as addressing objects by digest, some intra-repository references use *descriptors*. A descriptor augments a digest with extra descriptive information: media type, URLs, and strings (used in annotations), which are not modelled further.

$$[\textit{MediaType}, \textit{URL}, \textit{String}]$$

A media type is used to distinguish between certain types of objects in a repository. URLs indicate where certain objects are hosted in a network. Annotations provide arbitrary metadata for certain objects.

[1] collision resistant functions are designed so that it is extremely unlikely that distinct inputs produce the same output

A descriptor is a way of referring to an object in the same repository using a digest, a media type, a "raw" size, and zero or more URLs and annotations.

Descriptor

$$
\begin{aligned}
&\mathit{digest} : \mathit{Digest} \\
&\mathit{mediaType} : \mathit{MediaType} \\
&\mathit{size} : \mathbb{N} \\
&\mathit{urls} : \mathrm{seq}\ \mathit{URL} \\
&\mathit{annotations} : \mathit{String} \pfun \mathit{String}
\end{aligned}
$$

We will model a repository as a collection of content addressable objects, which we will introduce in turn, along with their media types. These objects are change sets, image configurations, image manifests, and image indices.

3 Change Sets ("Layers")

A changeset, or "layer", is a means of creating a root file system for a container based on an image.

File system paths and files (including directories) are modelled in terms of their usage, but not their implementation.

$$[FSPath, File]$$

We introduce a simple model of a file system which will be sufficient to show how the file system of an image is built from layers.

A simplified file system is a collection of files indexed by file system path.

$$\begin{array}{|l} \textit{FS} \\ \hline fs : FSPath \pfun File \\ \hline \end{array}$$

No mention is made of the hierarchical nature of file system paths. File modes and attributes and symbolic and hard links are not modelled.

An empty file system has no files.

$$EmptyFS \mathrel{\widehat=} [FS \mid fs = \varnothing]$$

An *image layer file system changeset*, informally known as a "layer", describes a particular type of modification to a file system where some files are first of all removed and then files are added or replaced.

$$\begin{array}{|l} \textit{ChangeSet} \\ \hline remove : \mathbb{P}\, FSPath \\ add : FSPath \pfun File \\ \hline \end{array}$$

There is a media type for changesets which will be used in descriptors that reference changesets.

$$\boxed{LAYER : MediaType}$$

A changeset is applied to a file system by removing files according to the changeset and then adding (or replacing) file according to the changeset.

$$
\begin{array}{|l}
\textit{ChangeSetApply} \\
\Delta FS \\
c? : \textit{ChangeSet} \\
\hline
fs' = (c?.\textit{remove} \lhd fs) \oplus c?.\textit{add} \\
\end{array}
$$

Applying a changeset to an empty file system results in a file system containing of just the additions from the changeset. So a sequence of changesets can be applied in turn starting with an empty file system to produce the "root" file system for a particular *container*.

The remainder of this document does not concern itself with combining changesets, except that it will show how an image refers to a sequence of changesets which can, as we have just seen, be used to form the root file system for each container constructed from the image.

3.1 Image Repositories (part 1 of 4)

We can now start to describe image repositories in terms of change sets. Other objects in an image repository will be described later.

An image repository has a content addressable collection of change sets.

$$
\begin{array}{|l}
\textit{RepositoryChangeSets} \\
\textit{changesets} : \textit{CAS}[\textit{ChangeSet}] \\
\end{array}
$$

This will make more sense once we start to add other objects to the repository.

4 Image Configurations

Container configurations and (image history) logs are used by container configurations, but are not modelled further.

$$[ContainerCfg, Log]$$

There is a media type for image configurations which will be used in descriptors that reference image configurations.

$$CONFIG : MediaType$$

An image configuration refers to the changesets (or "layers", of which there must be at least one) of an image and records the container configuration and history of the image.

Config

$$changesets : \mathrm{seq}_1\, Id$$
$$ccfg : ContainerCfg$$
$$history : \mathrm{seq}\, Log$$

4.1 Image Repositories (part 2 of 4)

We now add configurations to our description of an image repository and see how configurations relate to change sets. Other objects in an image repository will be described later.

An image repository has a content addressable collection of change sets and configurations addressed by id.

RepositoryConfigsAndLayers

$$RepositoryChangeSets$$
$$configs : CAS[Config]$$
$$ids : \mathbb{P}\, Id$$
$$digests : \mathbb{P}\, Digest$$

$$\forall\, cfg : \mathrm{ran}\, configs.deref \bullet \mathrm{ran}\, cfg.changesets \subseteq \mathrm{dom}\, changesets.identify$$
$$\langle \mathrm{dom}\, changesets.identify, \mathrm{dom}\, configs.identify \rangle \ \mathsf{partition}\ ids$$
$$\exists\, other : \mathbb{P}\, Digest \bullet$$
$$\quad \langle \mathrm{dom}\, changesets.deref, \mathrm{dom}\, configs.deref, other \rangle \ \mathsf{partition}\ digests$$

All configurations refer to valid changeset ids.

Ids and digests are partitioned between configurations and changesets (and, in the case of digests, other objects still to be introduced). That is, each id and each digest refers to at most one object in content addressable storage.

The next step is to introduce the central object in a repository: the *image manifest*.

5 Image Manifests

The central object in an image repository is the *image manifest* which describes a single image.

Image manifests have schema versions, which are not modelled further.

$$[SchemaVersion]$$

There is a media type for image manifests which is used both in image manifests and in descriptors that reference them.

$$IMAGE\_MANIFEST : MediaType$$

An image manifest refers to an image configuration and at least one changeset (or "layer").

Manifest

$$\begin{aligned} &schemaVersion : SchemaVersion \\ &mediaType : MediaType \\ &config : Descriptor \\ &changesets : \mathrm{seq}_1\, Descriptor \\ &annotations : String \pfun String \\ \hline &mediaType = IMAGE\_MANIFEST \end{aligned}$$

5.1 Image Repositories (part 3 of 4)

We extend our model of an image repository to include image manifests.

$$
\begin{array}{|l}
\textit{RepositoryManifestsConfigsAndLayers} \\
\textit{RepositoryConfigsAndLayers} \\
\textit{manifests} : \textit{CAS}[\textit{Manifest}] \\
\hline
\textit{manifests.identify} = \varnothing \\
\\
\exists\, \textit{other} : \mathbb{P}\, \textit{Digest} \bullet \\
\quad \langle \mathrm{dom}\, \textit{changesets.deref}, \mathrm{dom}\, \textit{configs.deref}, \\
\qquad \mathrm{dom}\, \textit{manifests.deref}, \textit{other} \rangle \ \mathsf{partition}\ \textit{digests} \\
\\
\forall\, \textit{man} : \mathrm{ran}\, \textit{manifests.deref} \bullet \\
\quad \textit{man.mediaType} = \textit{IMAGE\_MANIFEST} \land \\
\quad \textit{man.config.mediaType} = \textit{CONFIG} \land \\
\quad \textit{man.config.digest} \in \mathrm{dom}\, \textit{configs.deref} \land \\
\quad \textit{man.config.size} = \textit{configs.size man.config.digest} \land \\
\quad (\forall\, \textit{cs} : \mathrm{ran}\, \textit{man.changesets} \bullet \\
\qquad \textit{cs.mediaType} = \textit{LAYER} \land \\
\qquad \textit{cs.digest} \in \mathrm{dom}\, \textit{changesets.deref} \land \\
\qquad \textit{cs.size} = \textit{changesets.size cs.digest}) \\
\end{array}
$$

Manifests do not have ids.

Also, content addresses in the form of digests are partitioned between changesets, configs, manifests, and other objects (soon to be introduced). That is, each digest refers to at most one object in content addressable storage.

Also, all manifests have an appropriate media type and valid configuration and layer descriptors.

6 Image Indices

To complete our description of image repositories, we add another object, the *image index*. The main purpose of an image index is to enable a repository to contain multiple versions of an image for multiple platforms.

Image indices use extended descriptors which include platform information, which is not modelled further.

$$[Platform]$$

There is a media type for image indices which is used both in image indices and in descriptors that reference them.

$$IMAGE\_INDEX : MediaType$$

A manifest descriptor is a descriptor with an additional property denoting a platform.

ManifestDescriptor

$$Descriptor$$
$$platform : Platform$$

$$mediaType \in \{IMAGE\_MANIFEST, IMAGE\_INDEX\}$$

A manifest descriptor may refer to image manifests or image indices.

An image index refers to zero or more image manifests and/or image indices.

Index

$$schemaVersion : SchemaVersion$$
$$mediaType : MediaType$$
$$manifests : \mathrm{seq}\ ManifestDescriptor$$
$$annotations : String \pfun String$$

$$mediaType = IMAGE\_INDEX$$

6.1 Image Repositories (part 4 of 4)

We can now complete our model of image repositories.

An image repository is a content addressable collection of change sets, configurations, image manifests, and image indices.

Repository

$$
\begin{array}{l}
RepositoryManifestsConfigsAndLayers \\
indices : CAS[Index] \\
\hline
indices.identify = \varnothing \\
\\
\langle \mathrm{dom}\, changesets.deref, \mathrm{dom}\, configs.deref, \\
\quad \mathrm{dom}\, manifests.deref, \mathrm{dom}\, indices.deref \rangle \ \mathsf{partition}\ digests \\
\\
\forall\, idx : \mathrm{ran}\, indices.deref \bullet \\
\quad idx.mediaType = IMAGE\_INDEX \land \\
\quad (\forall\, man : \mathrm{ran}\, idx.manifests \bullet \\
\qquad (man.digest \in \mathrm{dom}\, manifests.deref \land \\
\qquad man.size = manifests.size\ man.digest) \lor \\
\qquad (man.digest \in \mathrm{dom}\, indices.deref \land \\
\qquad man.size = indices.size\ man.digest)) \\
\\
\exists\, next : Index \leftrightarrow Index \mid \forall\, i, j : Index \mid i \mapsto j \in next \bullet \\
\quad (\exists\, m : \mathrm{ran}\, i.manifests \bullet indices.deref\ m.digest = j) \bullet \\
\qquad next^{+} \cap \mathrm{id}\, Index = \varnothing
\end{array}
$$

Image indices do not have ids.

Also, content addresses in the form of digests are partitioned between changesets, configs, manifests, and indices. That is, each digest refers to at most one object in content addressable storage.

Also, all image indices have an appropriate media type and valid manifest descriptors (referring to image manifests and/or other image indices).

Also, there are no cycles of image indices.

A References

[1] Various authors, *Docker Inc.*, `https://www.docker.com/`.

[2] Various authors, *Open Container Initiative*, `https://www.opencontainers.org/`.

[3] Glyn Normington, *Image Registries*, `https://github.com/glyn/riff-specs/raw/master/distribution/distribution.pdf`.

[4] Various authors, *OCI Image Format Specification*, `https://github.com/opencontainers/image-spec/blob/master/spec.md`.

[5] Mike Spivey, *The Z Manual*, `https://www.cs.umd.edu/mvz/handouts/z-manual.pdf`.

[6] Mike Spivey, *The fuzz type-checker for Z*, `https://bitbucket.org/Spivey/fuzz`.

B Z Notation

Numbers:

$\mathbb{N}$ `Natural numbers {0,1,...}`

Propositional logic and the schema calculus:

$\ldots \land \ldots$	`And`	$\langle\!\langle \ldots \rangle\!\rangle$	`Free type injection`
$\ldots \lor \ldots$	`Or`	$[\ldots]$	`Given sets`
$\ldots \Rightarrow \ldots$	`Implies`	$', ?, !, _0 \ldots _9$	`Schema decorations`
$\forall .. \mid .. \bullet ..$	`For all`	$\ldots \vdash \ldots$	`theorem`
$\exists .. \mid .. \bullet ..$	`There exists`	$\theta \ldots$	`Binding formation`
$\ldots \setminus \ldots$	`Hiding`	$\lambda \ldots$	`Function definition`
$\ldots \mathrel{\widehat=} \ldots$	`Schema definition`	$\mu \ldots$	`Mu-expression`
$\ldots == \ldots$	`Abbreviation`	$\Delta \ldots$	`State change`
$\ldots ::= \ldots \mid \ldots$	`Free type definition`	$\Xi \ldots$	`Invariant state change`

Sets and sequences:

$\{\ldots\}$	`Set`	$\ldots \setminus \ldots$	`Set difference`
$\{.. \mid .. \bullet ..\}$	`Set comprehension`	$\bigcup \ldots$	`Distributed union`
$\mathbb{P} \ldots$	`Set of subsets of`	$\# \ldots$	`Cardinality`
$\varnothing$	`Empty set`	$\ldots \subseteq \ldots$	`Subset`
$\ldots \times \ldots$	`Cartesian product`	$\ldots \subset \ldots$	`Proper subset`
$\ldots \in \ldots$	`Set membership`	$\ldots$ partition $\ldots$	`Set partition`
$\ldots \notin \ldots$	`Set non-membership`	seq	`Sequences`
$\ldots \cup \ldots$	`Union`	$\langle \ldots \rangle$	`Sequence`
$\ldots \cap \ldots$	`Intersection`	disjoint $\ldots$	`Disjoint sequence of sets`

Functions and relations:

$\ldots \leftrightarrow \ldots$	`Relation`	$\ldots ^*$	`Reflexive-transitive`
$\ldots \nrightarrow \ldots$	`Partial function`		`closure`
$\ldots \rightarrow \ldots$	`Total function`	$\ldots (\!\mid \ldots \mid\!)$	`Relational image`
$\ldots \rightarrowtail \ldots$	`Partial injection`	$\ldots \oplus \ldots$	`Functional overriding`
$\ldots \mathrel{>\!\!\!\rightarrow} \ldots$	`Injection`	$\ldots \lhd \ldots$	`Domain restriction`
$\mathrm{dom} \ldots$	`Domain`	$\ldots \rhd \ldots$	`Range restriction`
$\mathrm{ran} \ldots$	`Range`	$\ldots \mathrel{\lhd\!\!\!-} \ldots$	`Domain subtraction`
$\ldots \mapsto \ldots$	`maplet`	$\ldots \mathrel{\rhd\!\!\!-} \ldots$	`Range subtraction`
$\ldots ^\sim$	`Relational inverse`		

Axiomatic descriptions:

Declarations

Predicates

Schema definitions:

SchemaName

Declaration

Predicates

Decorations:

(*undecorated*)	`Input state`	?	`Input parameter`
$'$	`Output state`	!	`Output parameter`